Advanced Core Data Usages

Session 143

These are confidential sessions—please refrain from filming or taking pictures.

Application Technologies

Advanced Core Data Usages

Ben Trumbull
Melissa Turner
Sr. Engineers, Core Data

http://developer.apple.com/wwdc2005/

Core Data Review

- Core Data is a model-driven object graph management and persistency framework
- Provides
  - Scalable object life cycle management
  - Automatic undo/redo support
  - User interface synchronization through Cocoa bindings integration
  - Data consistency and correctness
  - Reading/writing to various types of files

Core Data Benefits

- Relieves you from having to reinvent the wheel
- Faster development cycle
- Likely more robust applications
- Prepares your application for future features in Cocoa

What this Session Will Cover

- Architecture:
  The individual components
- The stack:
  Putting it all together
- Advanced topics:
  Working through the tough stuff
- Problems and decisions:
  Things to think about

Core Data Architecture

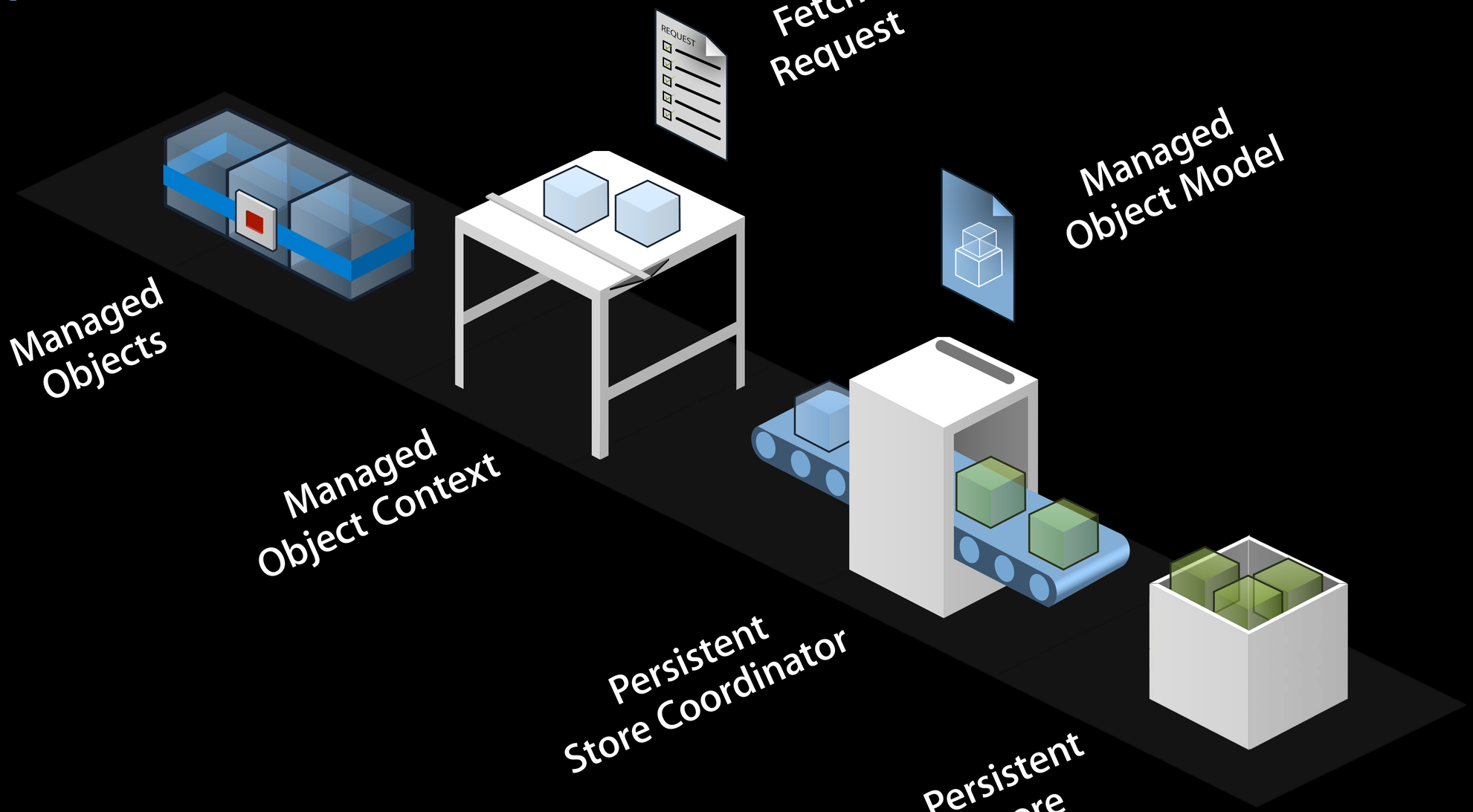
http://developer.apple.com/wwdc2005/

Architecture

- Core Data architecture built from the following
  - Models
  - Managed Objects and Object Context
  - Predicates and Fetch Requests
  - Object Stores and Store Coordinator
- Integration with Cocoa
  - Documents and Controllers
- Design tool in XCode can be used to create models
- XCode includes templates for creating Core Data applications

The Core Data Architecture Slide

The "persistence stack"



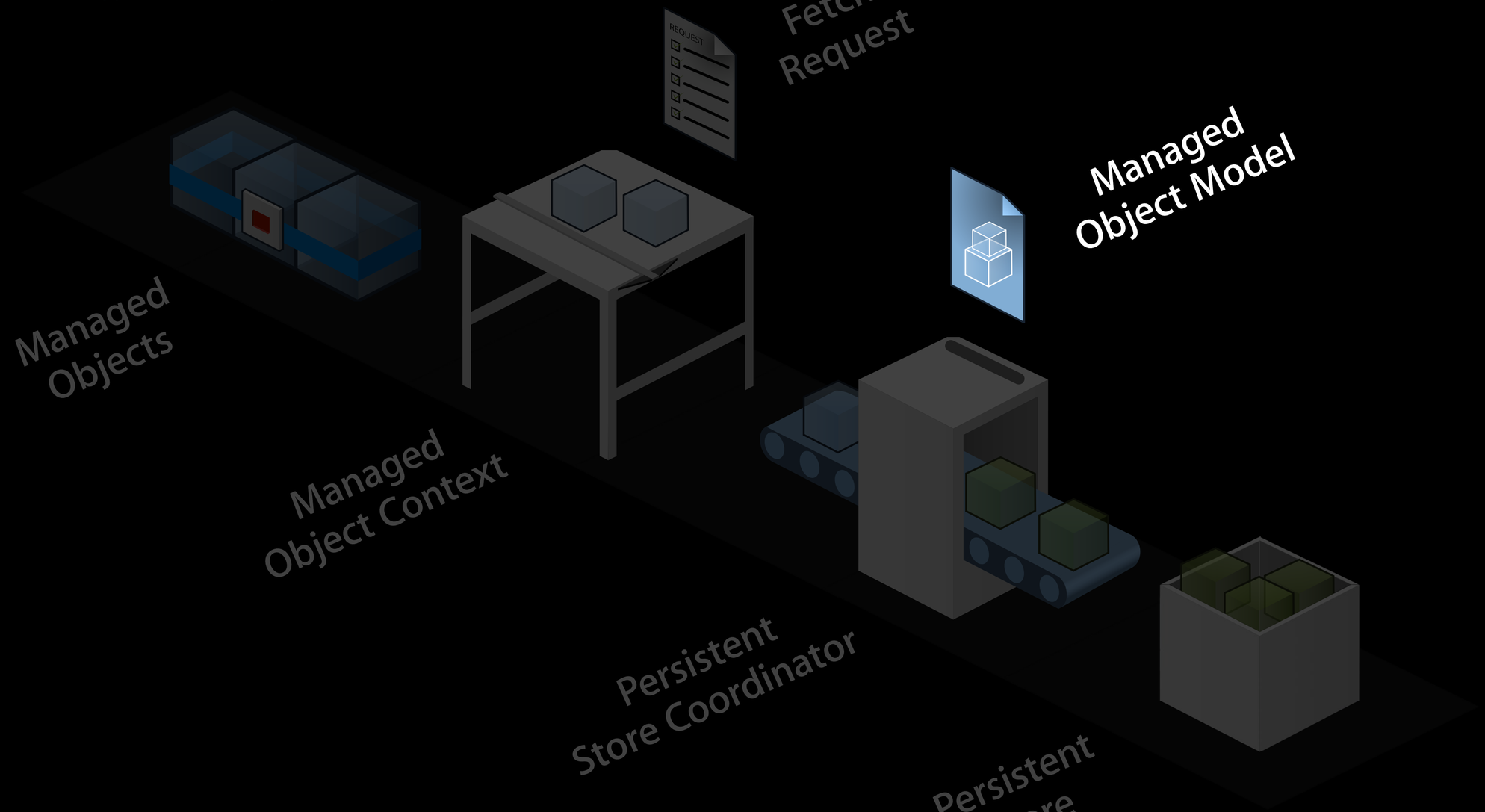
The Managed Object Model

NSManagedObjectModel

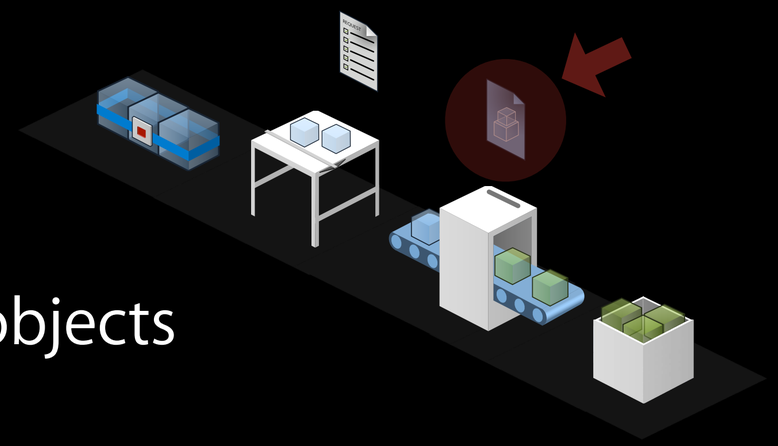
http://developer.apple.com/wwdc2005/

Core Data Models

NSManagedObjectModel

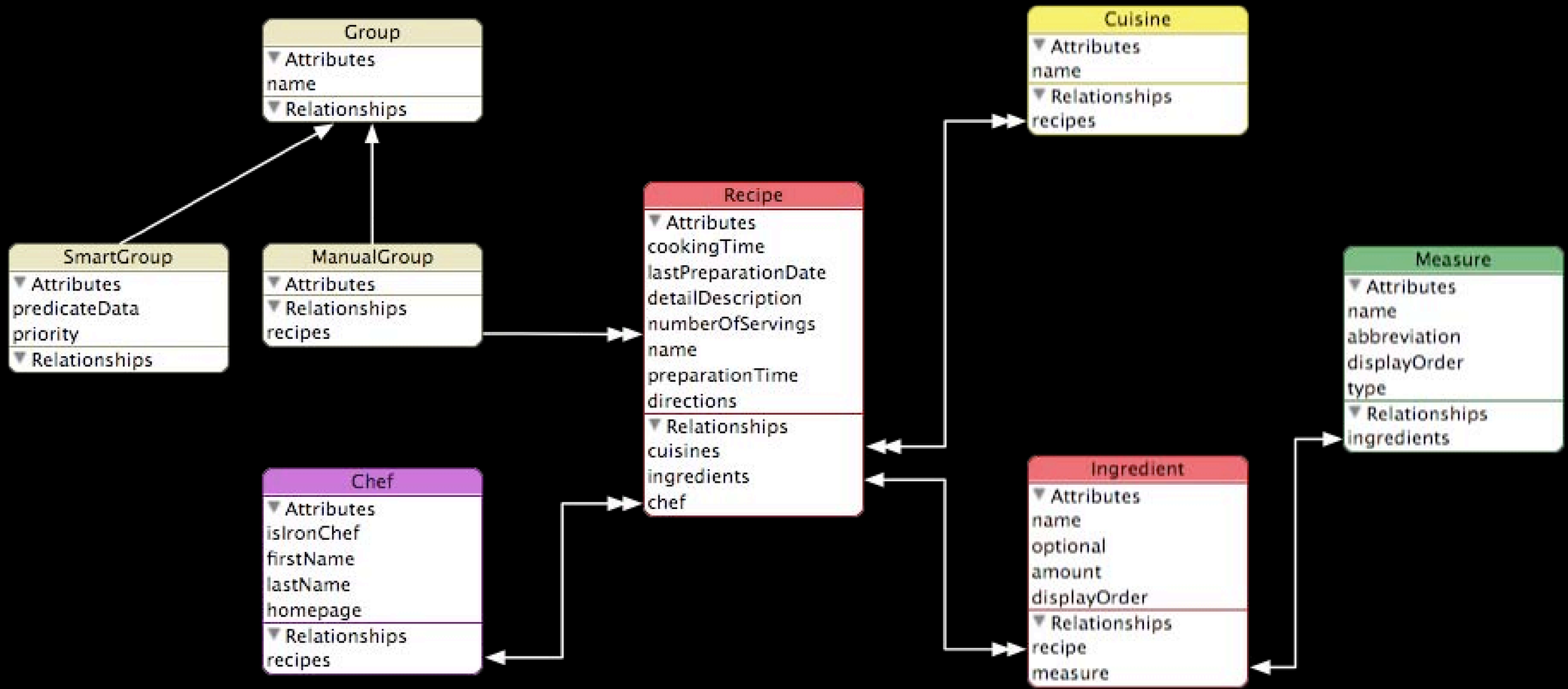


NSManagedObjectModel



- A model is a blueprint
  - Interaction between application data types and objects
  - Focus on architecture rather than flow
  - Baseline for model-based development
- Two main elements
  - Entity descriptions
  - Configurations

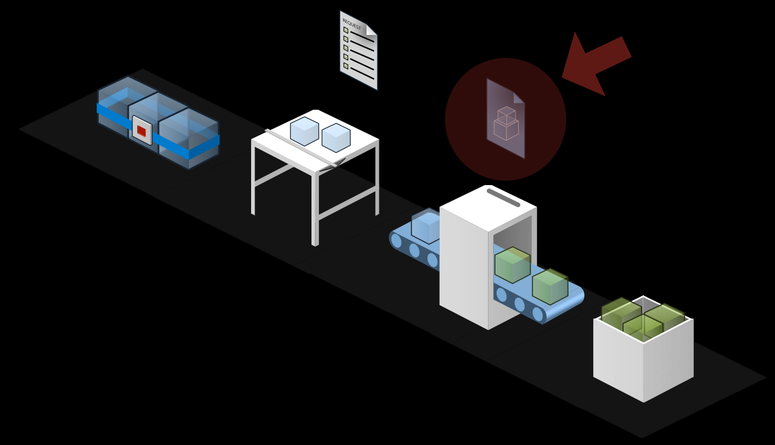
The Core Recipes Model



Anatomy of NSManagedObjectModel

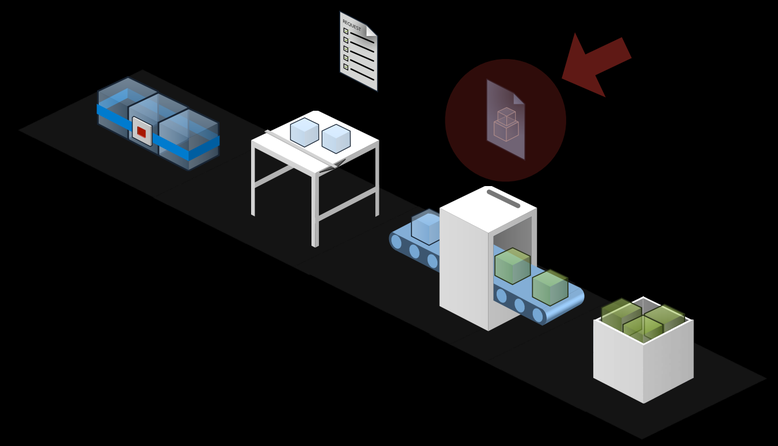
Element	Core Data Class
Entity	NSEntityDescription
Attribute	NSAttributeDescription
Relationship	NSRelationshipDescription
Fetched Property	NSFetchedProperty
Fetch Request Templates	NSFetchRequest
Configurations	n/a

Entities



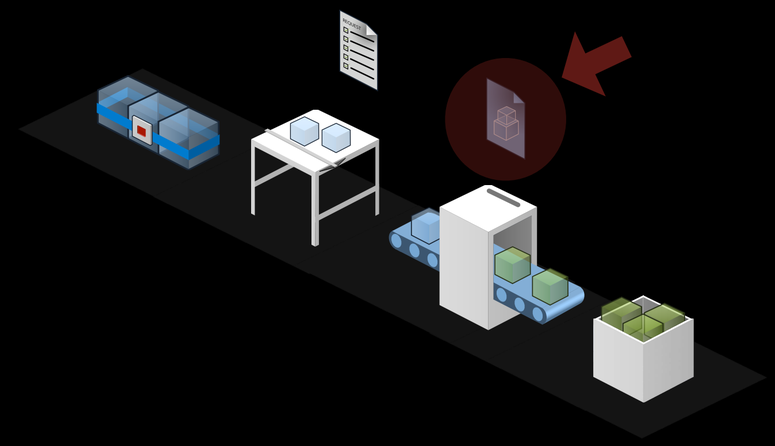
- Named bag containing information about related data
- Several property types
  - Attributes
    - Contain value data
  - Relationships
    - Contain interaction information
  - Fetched properties
    - Allow for weak unidirectional relationships
    - Special predicate variables
    - Only refresh when object is refreshed

Properties



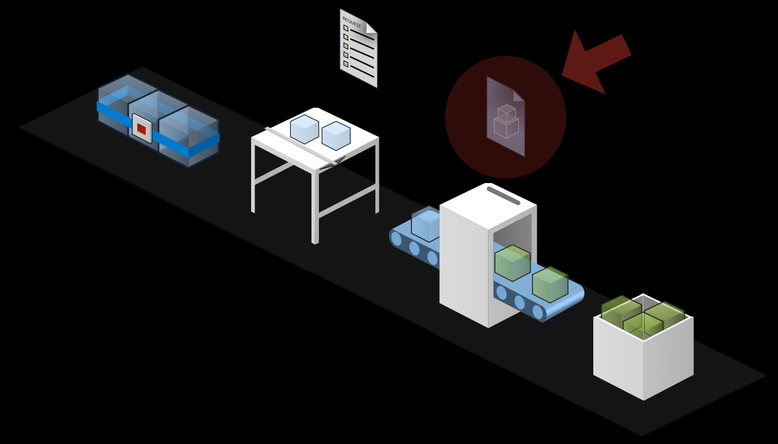
- Data containing elements of an entity
  - Can be transient or persistent
- Transient properties are used to
  - Store cached data calculated during application processing
  - Hold unsupported data types
  - Contain cross-store relationship information
  - Store derived properties

Relationships



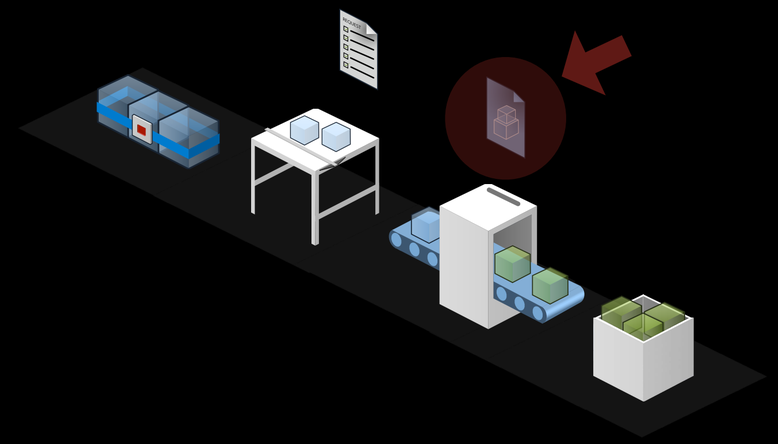
- Relationships are sets, not arrays
  - No ordering
  - Unique correlations
- Core Data manages inverse relationships for you
- Delete rules
  - NSNullifyDeleteRule
  - NSCascadeDeleteRule
  - NSDenyDeleteRule
  - NSNoActionDeleteRule

Relationships Are Not Ordered



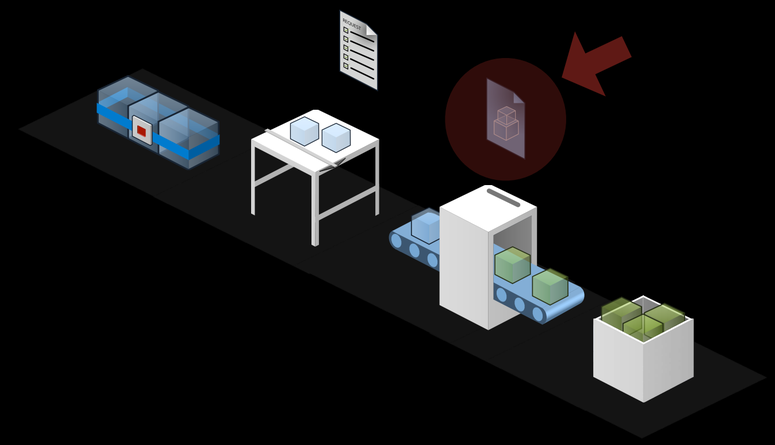
- Most data are not inherently ordered
  - Employees
  - Books
- Some data are ordered
  - Entries in a ledger
- Do you really need order? Or merely the appearance of order?

Problem: Achieving Order



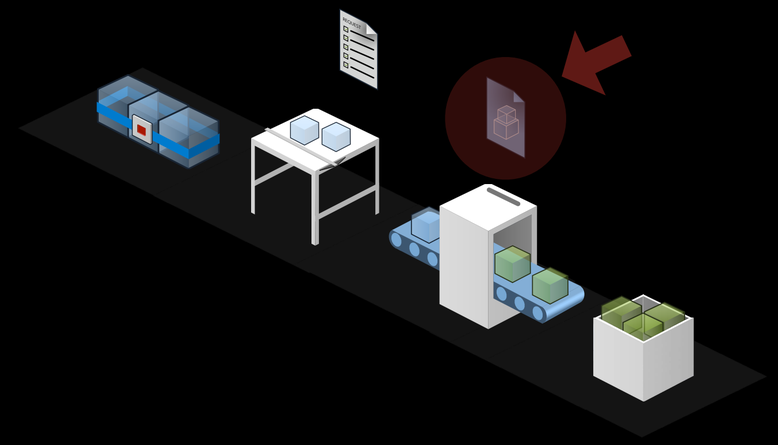
- Appearance of order
  - Use fetch requests with sort orderings
  - Use sort orderings on array controllers
  - Use NSArray's sorting methods
- For real order
  - Create an order column attribute
    - You'll have to maintain this explicitly

Loading Models



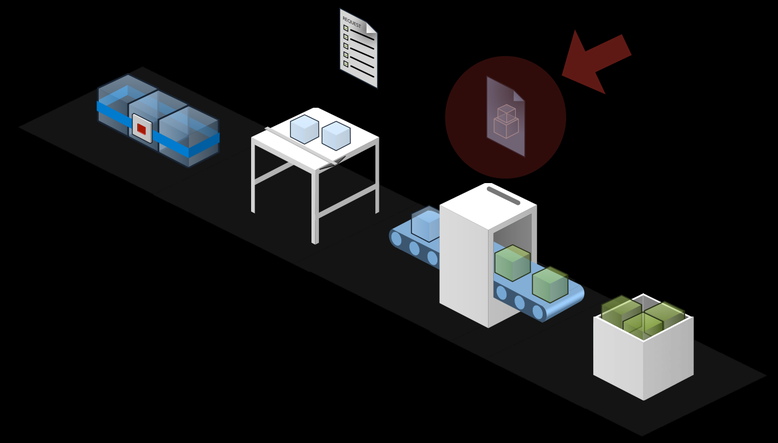
- A single model
  - -initWithContentsOfURL:
- Multiple models from multiple bundles can be merged at runtime
  - +mergedModelFromBundles:
  - +modelByMergingModels:
- No entity overlap
- Configurations will be merged as well

Problem: Good Model Design



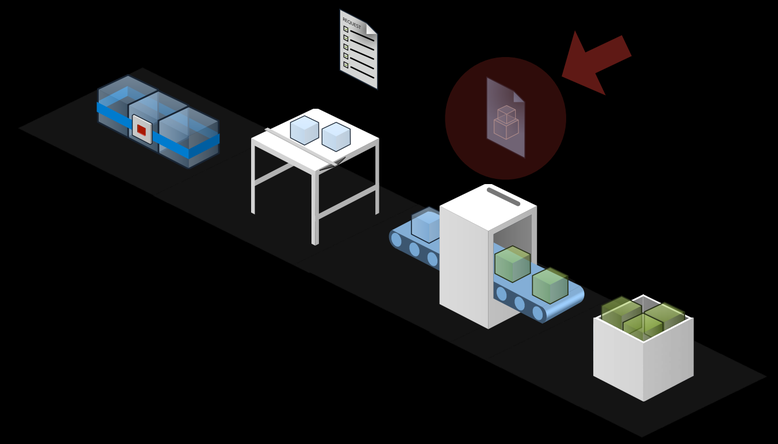
- Put large data objects at the end of one-to-one relationships
- Can’t query NSData attribute types
  - Consider extracting bits you may want to query
- Normalize dataset to avoid duplication or loading more data than necessary
- Caveat: Joins are expensive
  - Consider denormalizing data if frequently used queries will use joins

Problem: Old Data, New Model

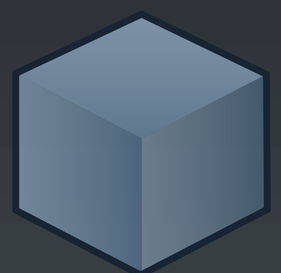


- Applications change over time
- Users have old data they want to use
- Generic solution is difficult
  - Entity inheritance
  - Relationships
  - Dependent properties
- How to upgrade stores
  - Create two stacks, one with each model
  - Load objects from old store; copy data into new objects in new store

Store Upgrade Tips and Tricks



- Multiple passes over data
  - Move discrete, separable chunks independently
  - Minimize memory use
- Models can be modified at runtime
  - Temporary disabling of validation requirements
  - Use NSManagedObject instead of custom classes
- See upgrade example on WWDC source CD



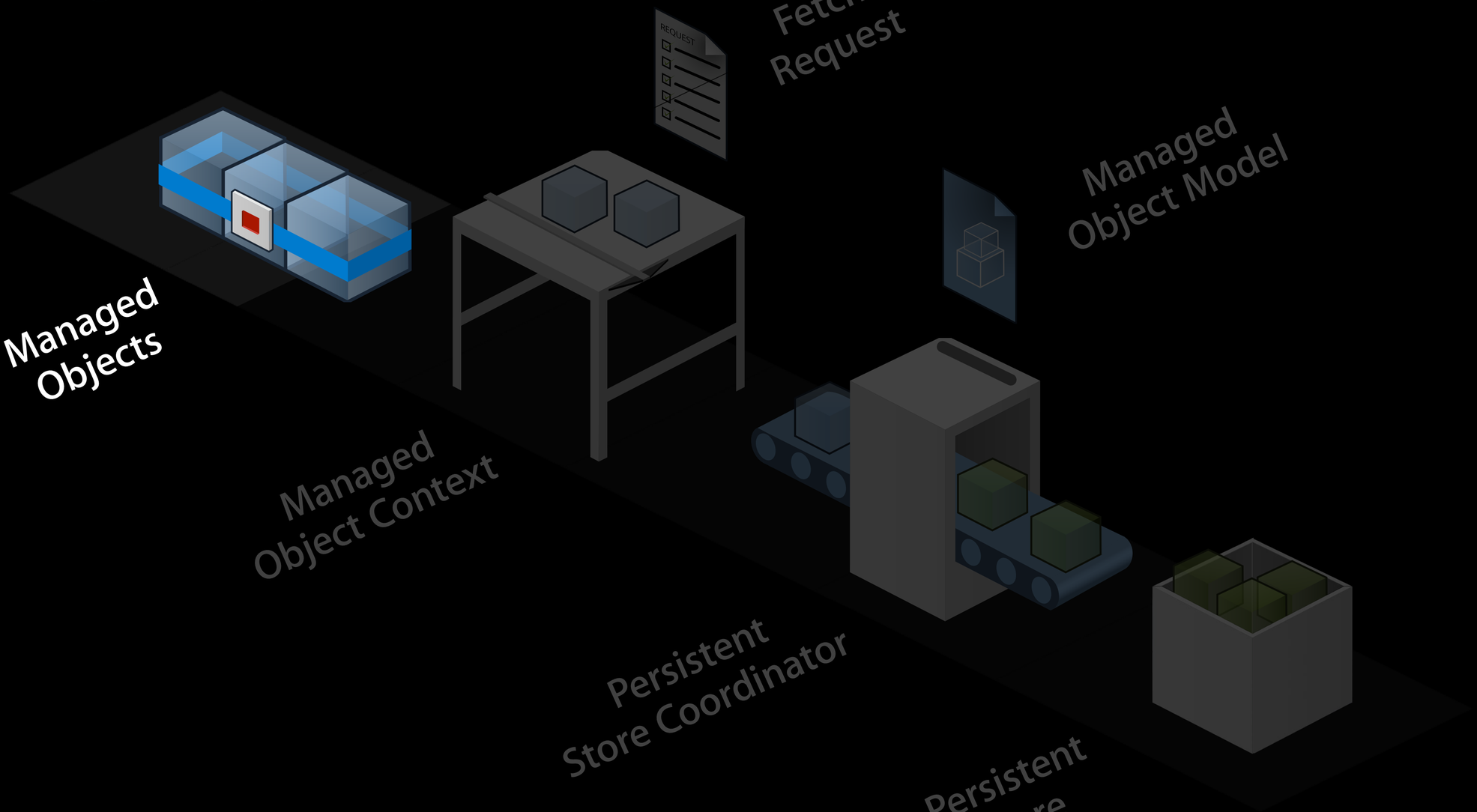
The Managed Object

NSManagedObject

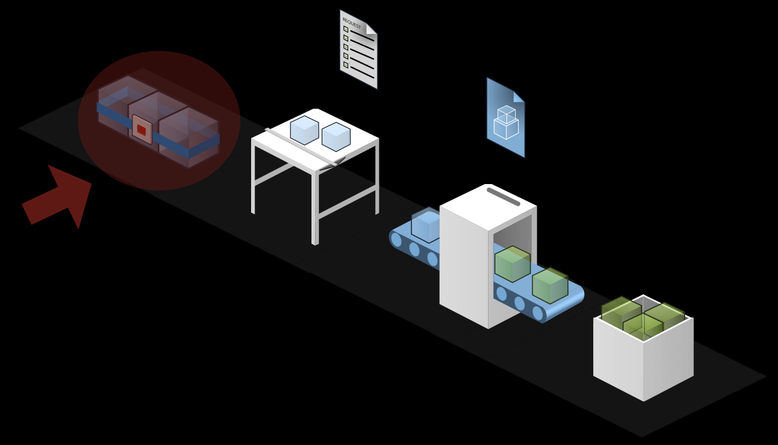
http://developer.apple.com/wwdc2005/

Core Data: Managed Objects

NSManagedObject

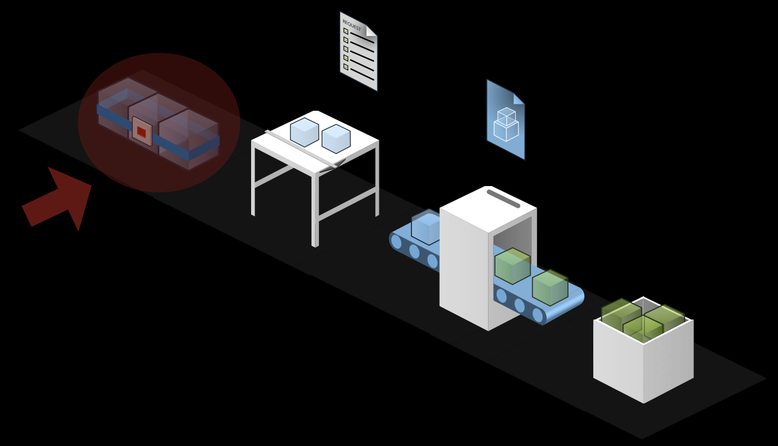


Managed Objects



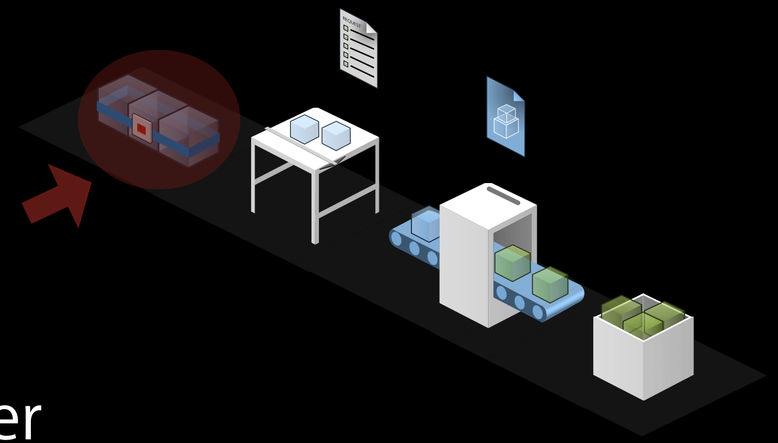
- NSManagedObject is the root class for Core Data
- Managed Objects are to Entity as instances are to Class
- Basic unit for development and operations
  - Fetch, insert, update, delete, save
  - Analogous to a database row or XML node
- Core Data noun

Managed Objects



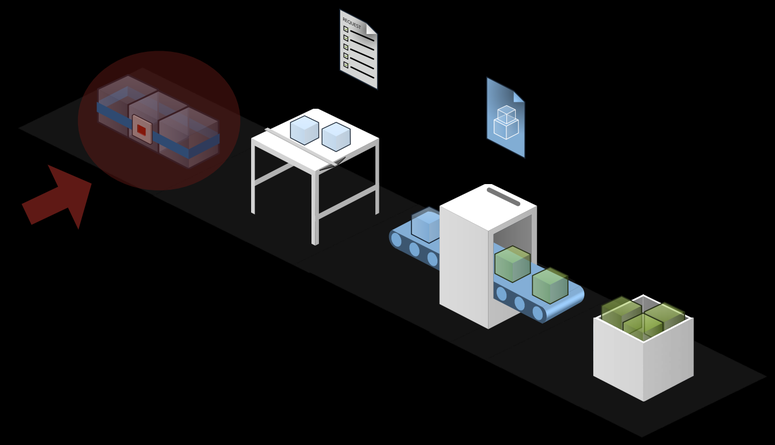
- Each Managed Object
  - Described by a single entity
  - Possesses unique NSManagedObjectID
  - Associated with one NSManagedObjectContext
- Subclasses can implement
  - Custom logic
  - Framework callbacks
  - Validation
- Conforms to NSKeyValueCoding
- Conforms to NSKeyValueObserving

Lifecycle of a Managed Object



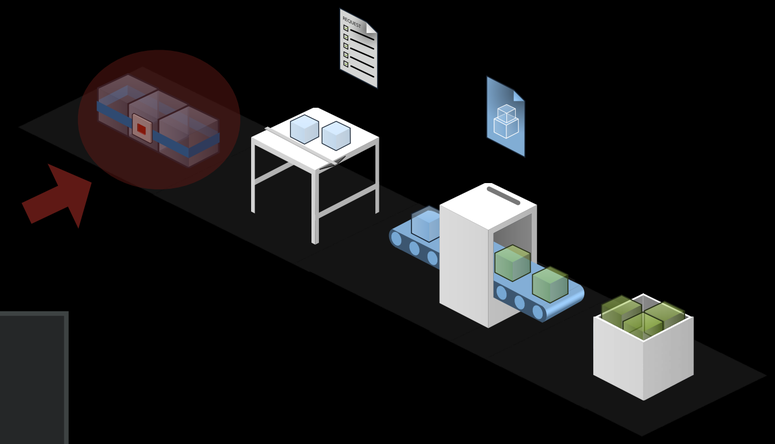
- Normal object lifecycle
  - alloc/init/retain/release/dealloc
  - Once an object is deallocated, it's data is no longer accessible
- Persistent object lifecycle
  - initWithEntity:inManagedObjectContext: & save
  - deleteObject: & save
  - retain/release/dealloc works normally for the current working set
  - dealloc does not cause a persistent object to be deleted

KVC/KVO Support



- Core Data provides KVC and KVO over all modeled properties
- mutableSetValueForKey:
- Dependent key notifications
- KVC and setValue:forKey: retains, does not copy
  - Don’t set a property to a mutable object and then mutate that object

Primitive KeyValueCoding

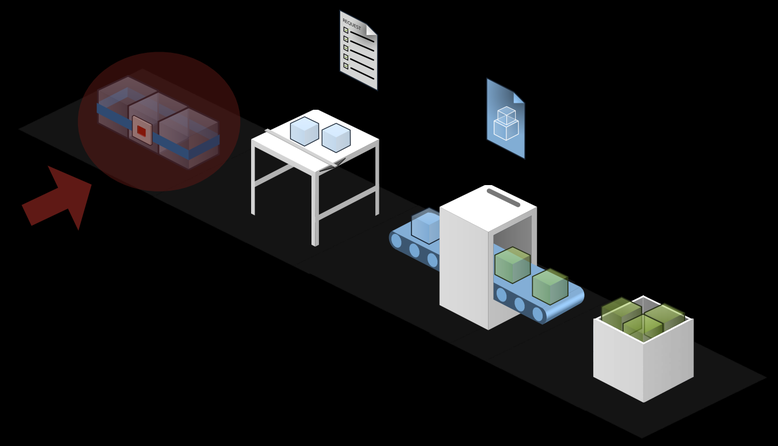


- NSManagedObject provides API:

```
- (id)primitiveValueForKey:(NSString*)key;
- (void)setPrimitiveValue:(id)value forKey:(NSString*)key;
```

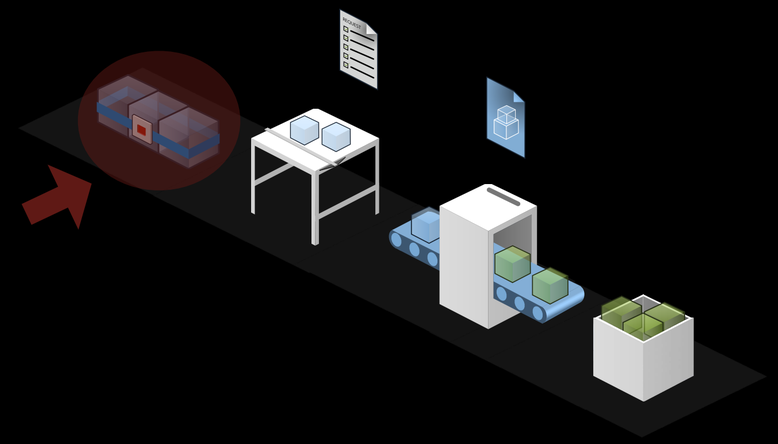
- Does **NOT** go through KVC accessor methods
- Does **NOT** perform validation, argument checking, or relationship integrity management
- **ONLY** for subclasses to implement custom accessor methods
- What does this mean for sets (ie setPrimitiveValueForKey: mySet)

NSManagedObjectID



- Identifies managed objects in contexts or applications
- Managed objects with the same ID refer to the same
  - XML node
  - SQL row
- Managed object IDs can be archived and recreated later
- Offer a URI representation

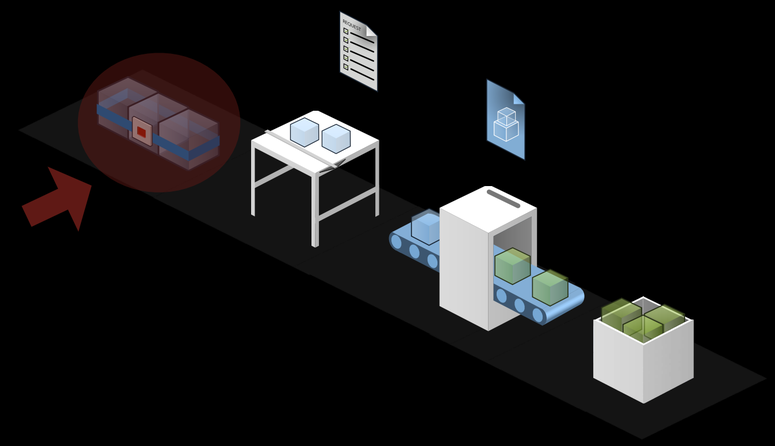
Object IDs



- NSManagedObjectIDs unique to their persistent store
- Customizing the URI scheme
- Passing URIs to other applications
  - objectWithID + executeFetchRequest (self = ID)
  - Issues with temporary IDs
    - Permanent ids assigned during save
- Object IDs can be passed around in a thread safe manner
- IDs change for newly inserted objects and store migrations

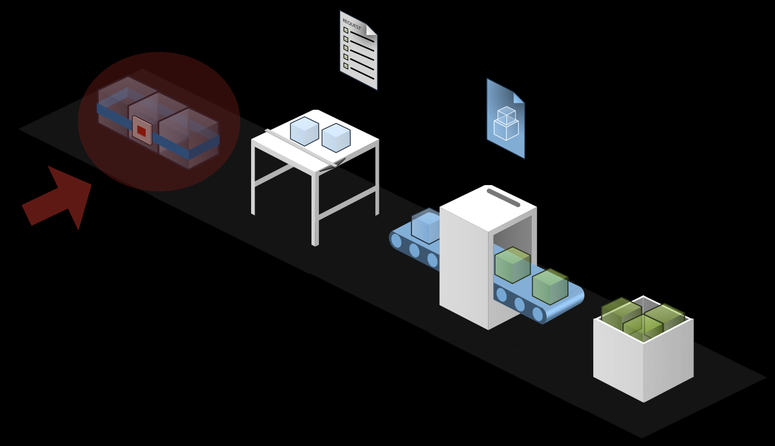
Subclassing NSManagedObject

Subclassing



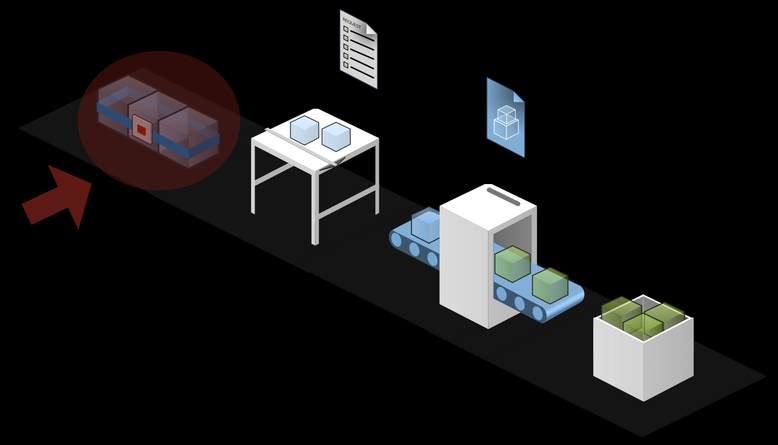
- Life cycle of NSManagedObjects belongs to framework
  - Only need to initialize objects you are inserting
- Don’t use -init
  - Managed Objects without entities are very unhappy
  - Use initWithEntity:insertIntoManagedObjectContext:
- Cleanup in didTurnIntoFault instead of dealloc

Subclassing NSManagedObject



- Callbacks
  - -awakeFromFetch
  - -awakeFromInsert
  - -didTurnIntoFault
  - -willSave
  - -didSave
- Validation callbacks
  - -validationForDelete:
  - -validateForInsert:
  - -validateForUpdate:

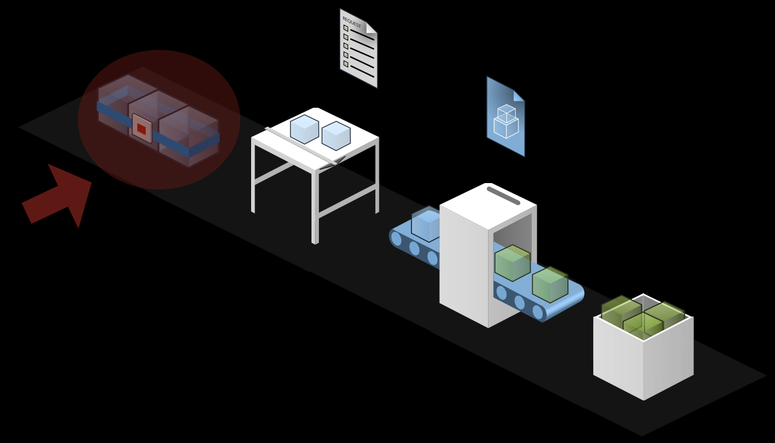
Let Us Manage Space for Properties



- More efficient than defining your own ivars
- Encouraged for attributes
- Mandatory for to-many relationships and fetched properties
- For to-many relationships:
  - Prefer -mutableSetValueForKey: to custom accessor methods
    - Much more efficient

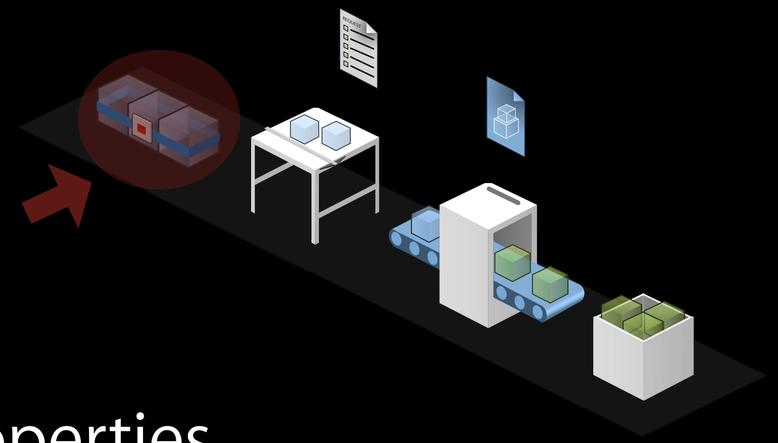
Using ivars for Modeled Attributes

If you must

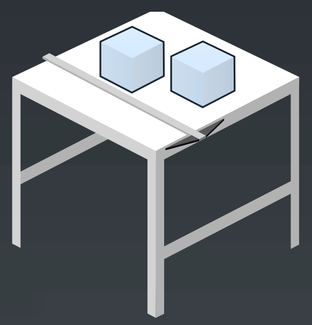


- Attributes should use immutable value classes
  - NSString, not NSMutableString
  - Scalar types supported by KVC are also okay
- Use accessor methods which send KVO notifications

Unmodeled Properties



- Data not described by the Entity
- Usually ivars on your subclass
- Core Data does not interact with unmodeled properties
  - No persistence
  - No undo
- Free form customization
- Maintain in same callbacks as transient properties



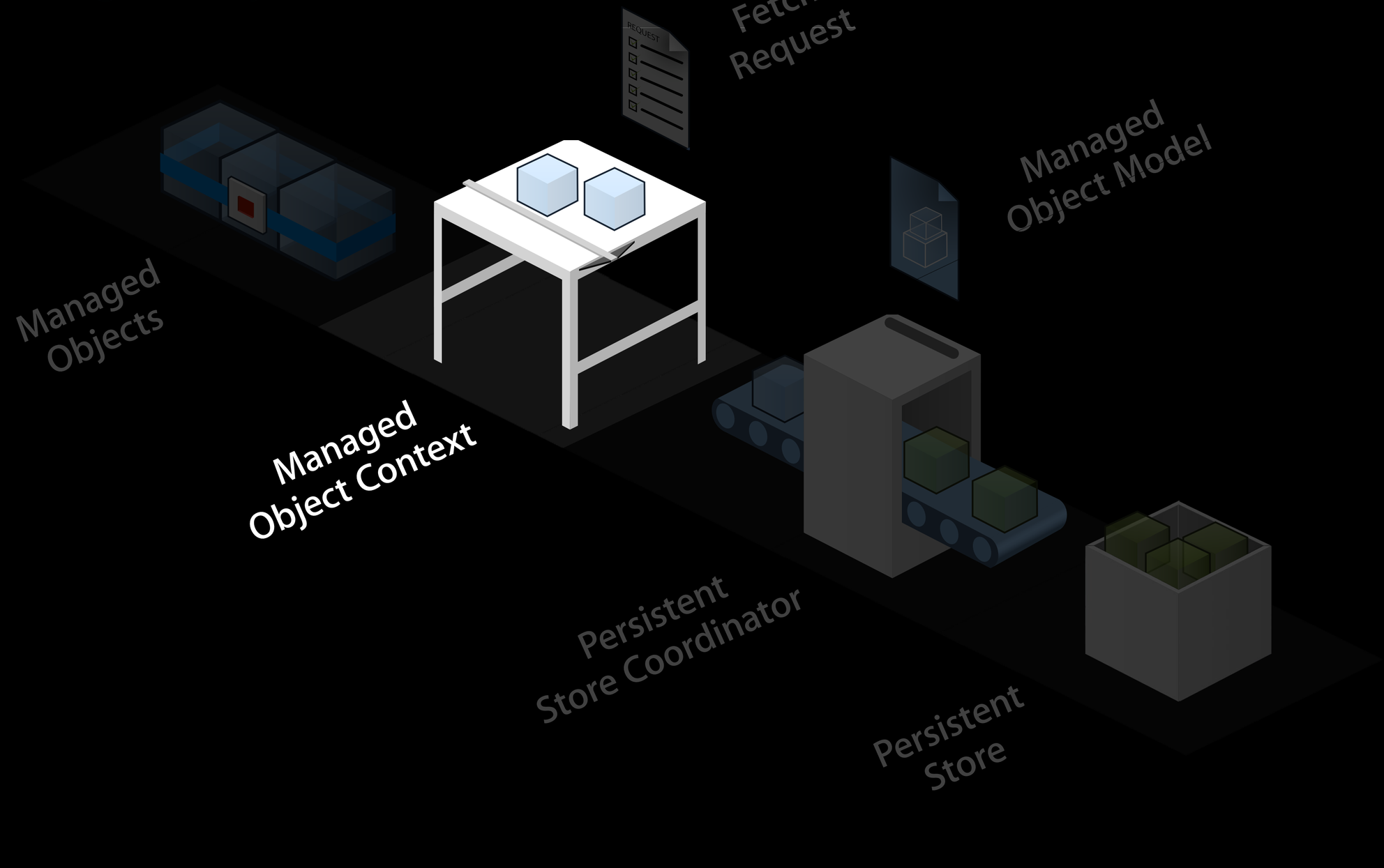
The Managed Object Context

NSManagedObjectContext

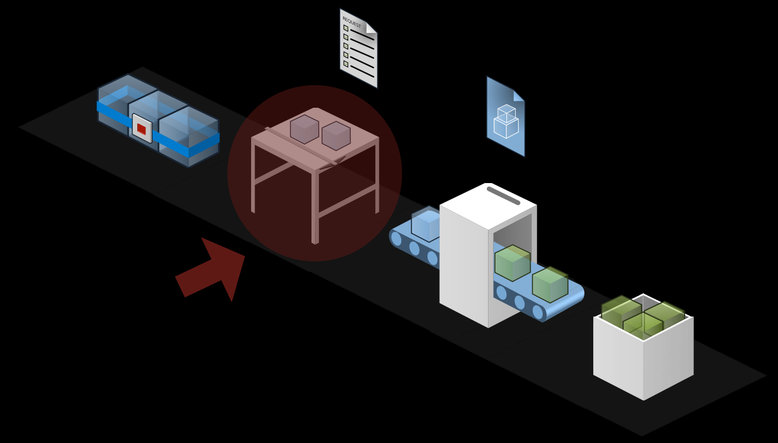
http://developer.apple.com/wwdc2005/

Core Data: Managed Object Context

NSManagedObjectContext

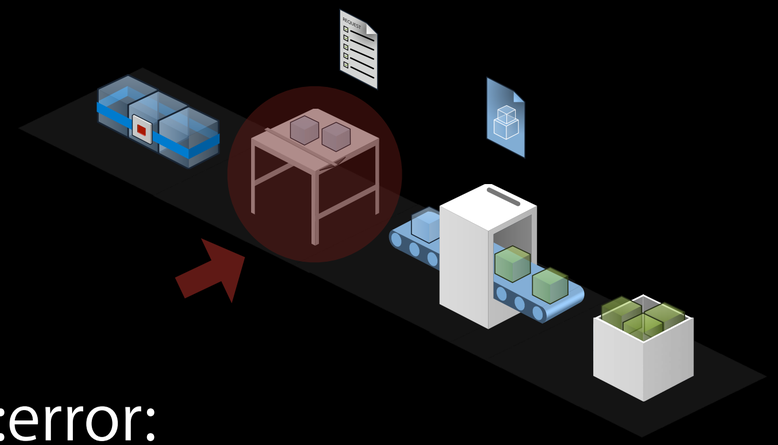


Managed Object Context



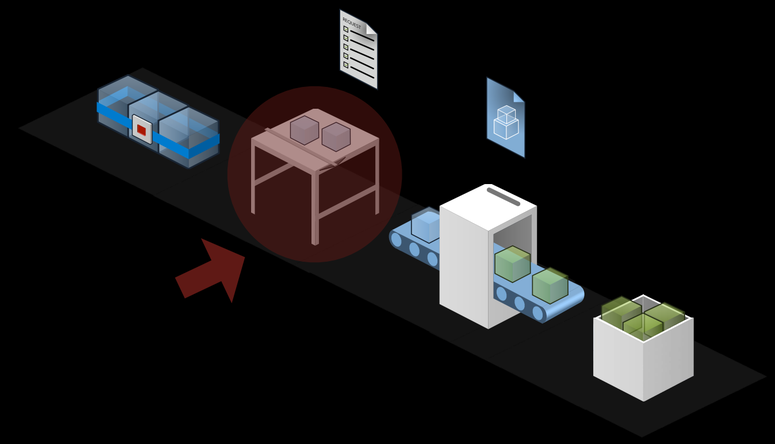
- Working set of Managed Objects
- Scratch pad for changes
- Responsibilities
  - Fetches objects from external store
  - Saves and reverts changes
  - Undo and redo operations
  - Notifications and callbacks
- Core Data verbs

Finding Data



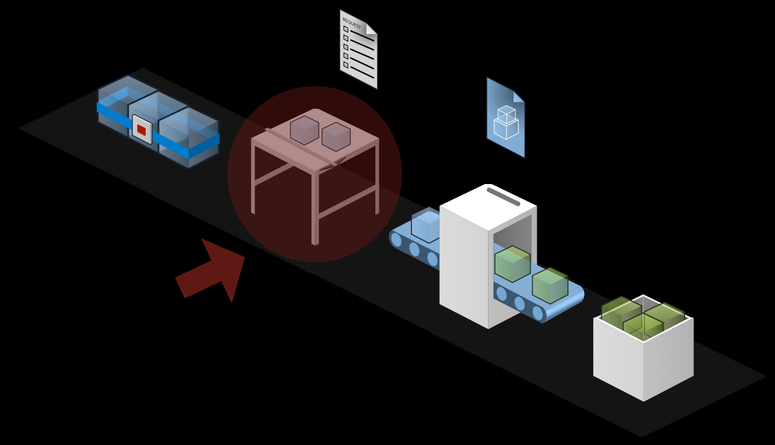
- Fetch requests
  - Entity, Predicate, Sort descriptor
  - Pass request to Context via executeFetchRequest:error:
- By object ID
  - -objectWithID:
    - Always return the object or a proxy for it
  - -objectRegisteredForID:
    - Returns nil if the object has not already been fetched or created

Unsaved Changes Affect Fetching



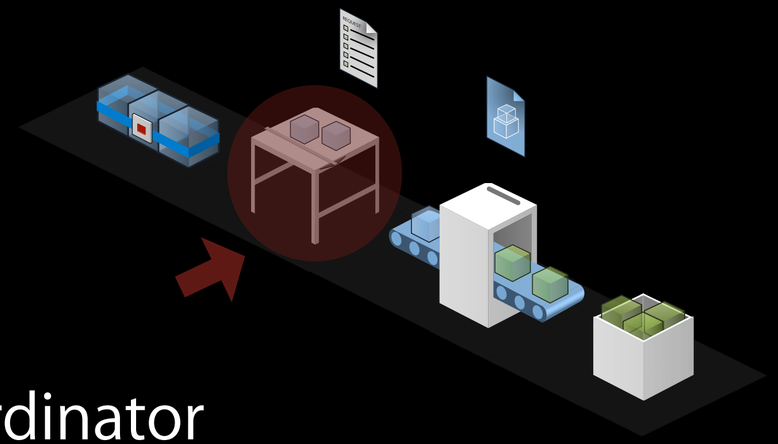
- View of your data incorporating some pending changes
- Fetch results reflect those changes
  - If changed objects match the fetch request's predicate
  - Inserts added
  - Deletions removed
  - Changes merged
- Fetching with lots of unsaved changes slows performance

Deleting an Object from a MOC



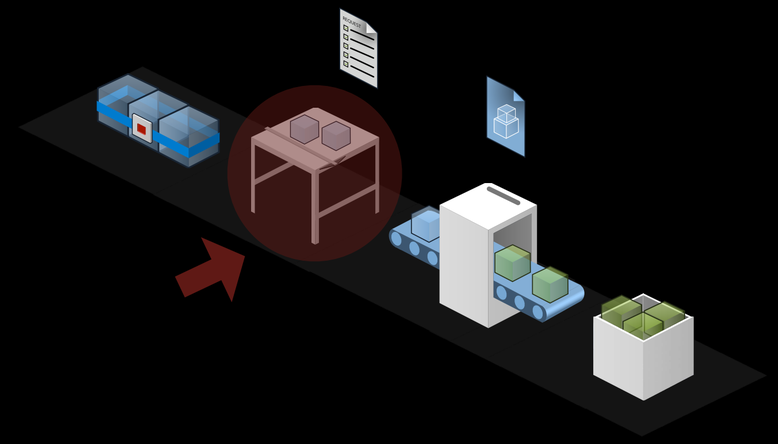
- Send `-deleteObject:` to the context
- Deletes propagate through relationships
  - Propagation is delayed
- Until `-saveChanges:` invoked
  - Changes are coalesced and remain pending
  - Objects are valid and usable
  - Deletion can be undone

Saving Changes in a MOC



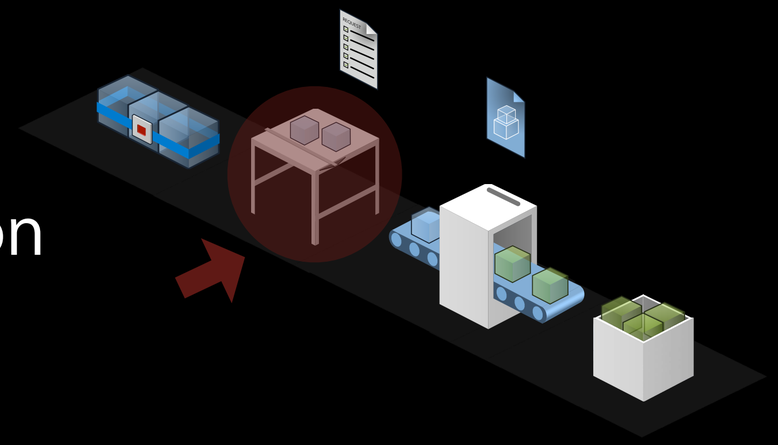
- Managed Object Context
  - Groups all the insertions, deletions, and updates
  - Passes a save request to the Persistent Store Coordinator
- Persistent Store Coordinator
  - Sends changes to the appropriate object store
- Each persistent store
  - Writes out the changes using its native mechanism

Reverting Changes in a MOC



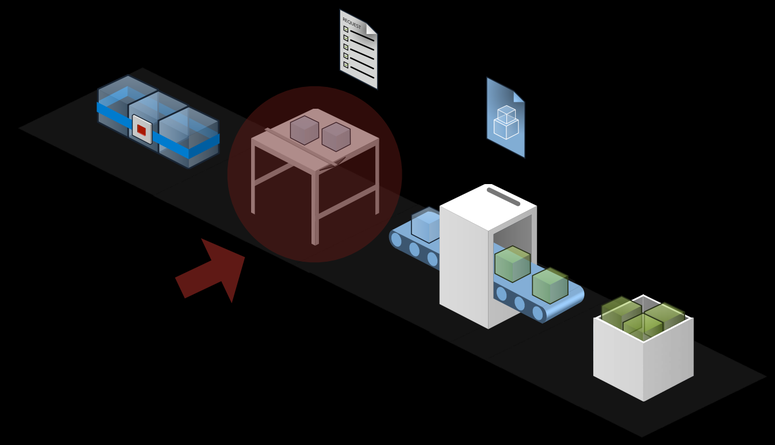
- Rollback (aka revert)
  - Returns to the last saved state
- Reset (clean slate entirely)
  - Invalidates all the objects with this context
  - Discard references to those objects and fetch them again
- Supports NSUndoManager
  - Undo
  - Redo

Undo and Redo



- Core Data uses NSUndoManager from Foundation
- Event loop based groupings
  - Changes coalesced each event loop
  - processPendingChanges
- Undo/redo across saves
- Disabling undo

Synchronizing Changes

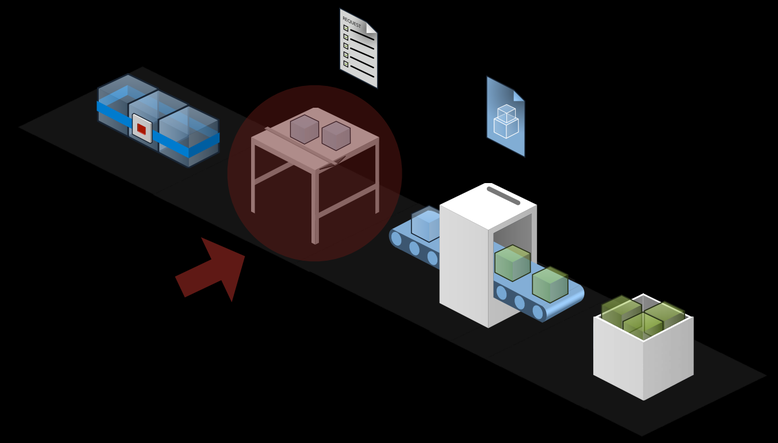


- Data freshness
  - Keeping objects up to date
  - Keeping similar objects in sync between different contexts or applications
- -refreshObject:mergeChanges:
- -setStalenessInterval:
- Alternatively, allow data to get stale
  - Handle conflicts with merge policies

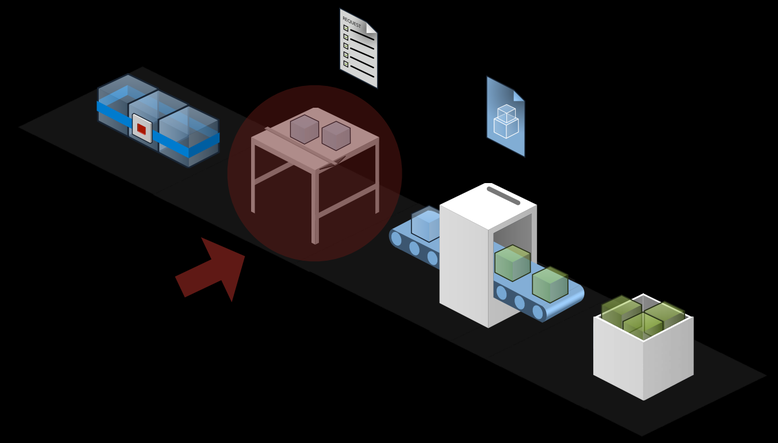
Merge Policies

Conflicting Writes

- Multiple writers
  - Multiple threads
  - One thread juggling multiple contexts
  - sqlite store and multiple applications

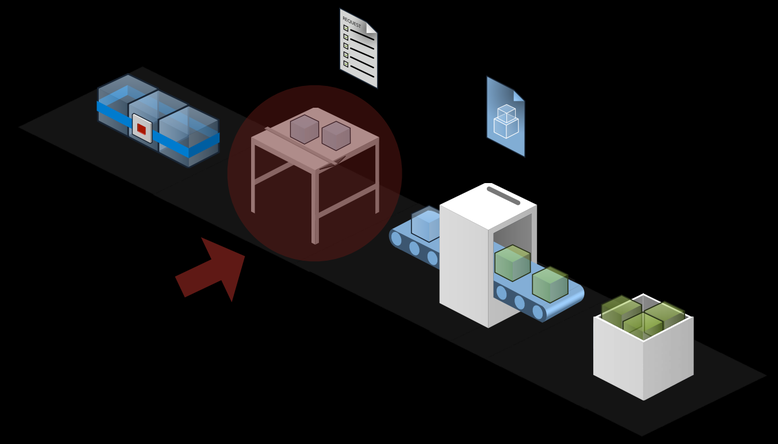


Detecting Conflicts



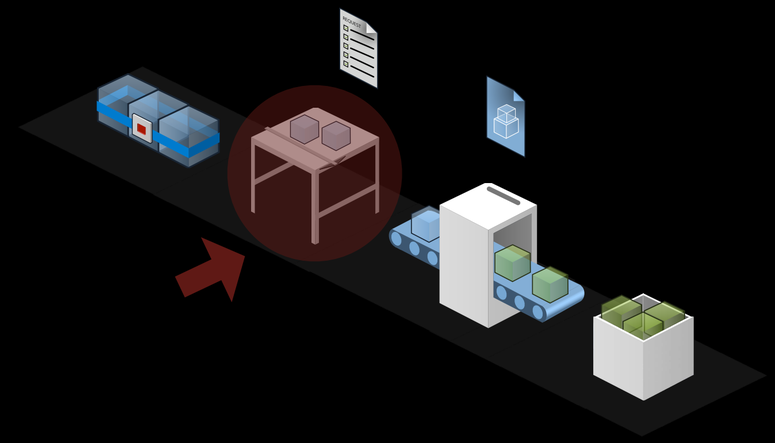
- Optimistic locking
  - Core Data assumes saves will succeed
  - Rolls back if a conflict occurs
- Only sqlite store handles conflicts on disk
  - XML and Binary stores perform atomic reads and writes

Merge Policies



- NSErrorMergePolicy
  - Default
  - userInfo has details for each conflict
- NSOverwriteMergePolicy
  - Last writer wins
- NSRollbackMergePolicy
  - First writer wins
- NSMergeByPropertyObjectTrumpMergePolicy
  - On property by property basis; last writer wins
- NSMergeByPropertyStoreTrumpMergePolicy
  - On property by property basis; first writer wins

Conflicts Involving Unchanged Data



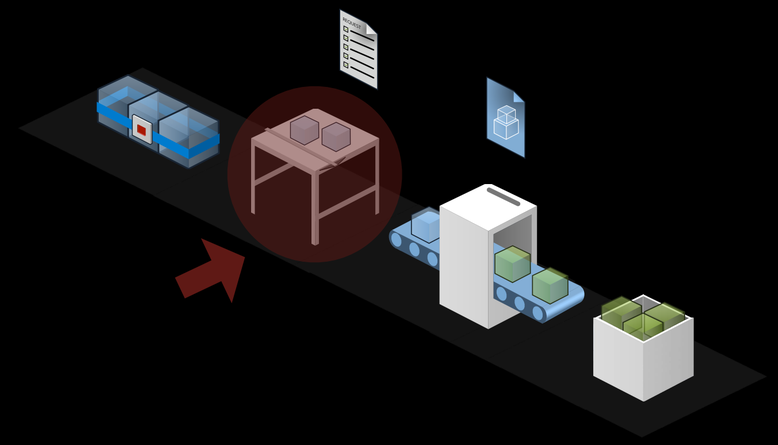
- Semantic conflicts
- -detectConflictsForObject:
  - RunningTotal = Cost1 + Cost2
  - Cost1 and Cost2 are properties on unchanged objects
  - Save new RunningTotal
    - But fail if Cost1 or Cost2 changed externally

Working With Multiple Contexts

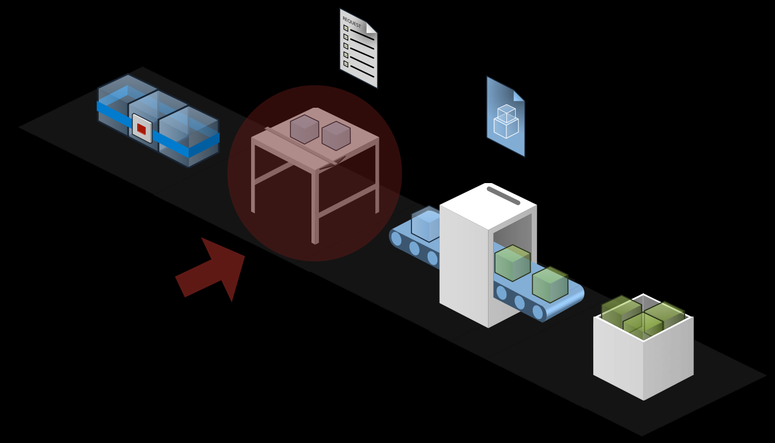
http://developer.apple.com/wwdc2005/

Why Do it?

- Isolate changes in different editors
- Save or undo different groups of changes independently
- Inspector windows
- Reduce working set in each context
- Work with multiple Persistent Store Coordinators

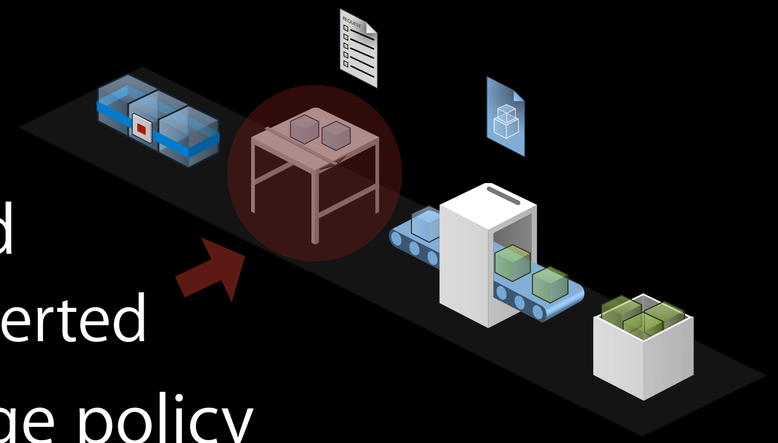


Things to Do

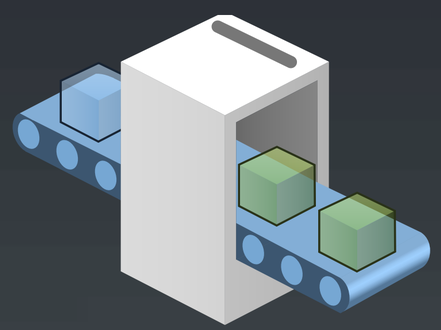


- Pass objectIDs around
  - Managed Objects can only be associated with one context
- If contexts are using the same coordinator
  - Load the manage object with objectWithID:
  - Or perform a fetch with a predicate
    - `[NSPredicate predicateWithFormat:@"self = %@", objectid]`
    - `[NSPredicate predicateWithFormat:@"self IN %@", arrayofids]`
- If contexts are using different coordinators
  - Persistent store must be added to both coordinators

Details



- Newly inserted objects are temporary until saved
  - Visible only in the context in which they were inserted
- Contexts changing the same data will need merge policy
- Sharing Persistent Store Coordinator shares caches
- Multiple Persistent Store Coordinator duplicates caches



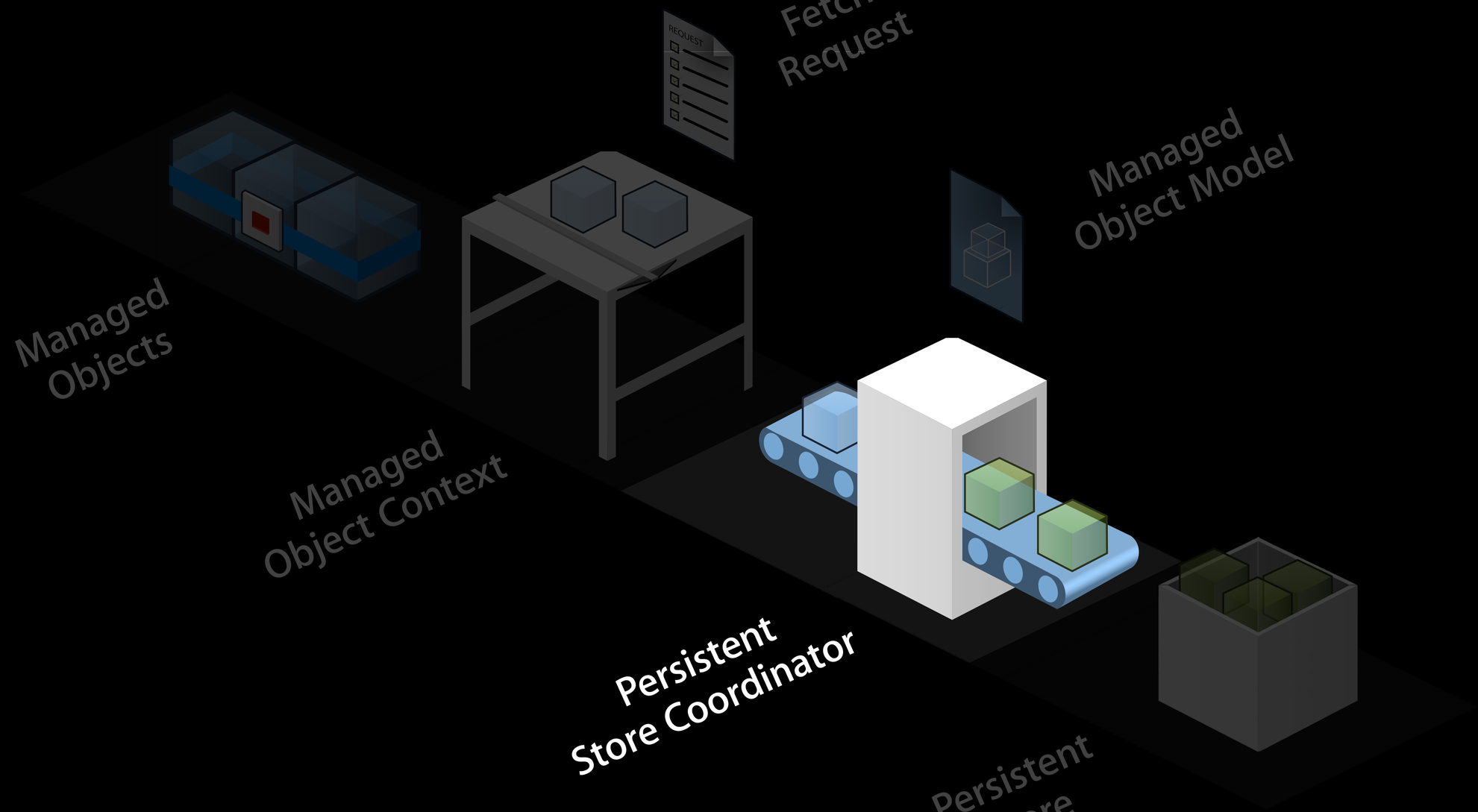
The Persistent Store Coordinator

And object stores

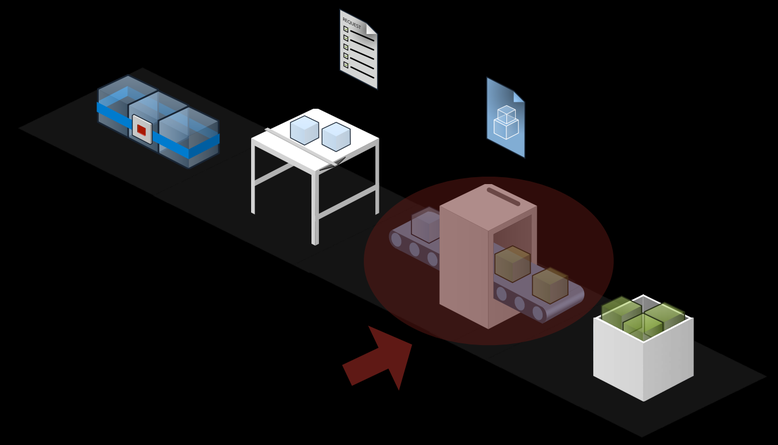
http://developer.apple.com/wwdc2005/

Core Data: Persistent Store Coordinator

NSPersistentStoreCoordinator

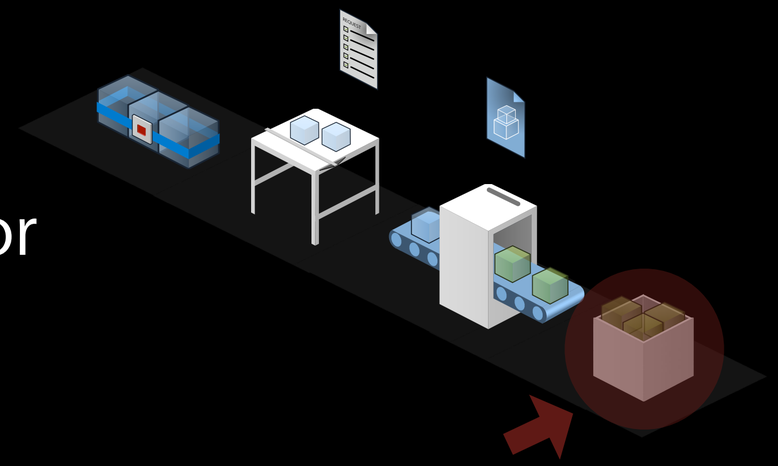


NSPersistentStoreCoordinator



- Bridge between object lifecycle management and persistence mechanisms
- Center of the framework's world
  - One persistent store coordinator per stack
  - Many contexts, many stores
- Provides a facade of a single store to the MOC
- Initialized with a managed object model

What, No NSObjectStore?



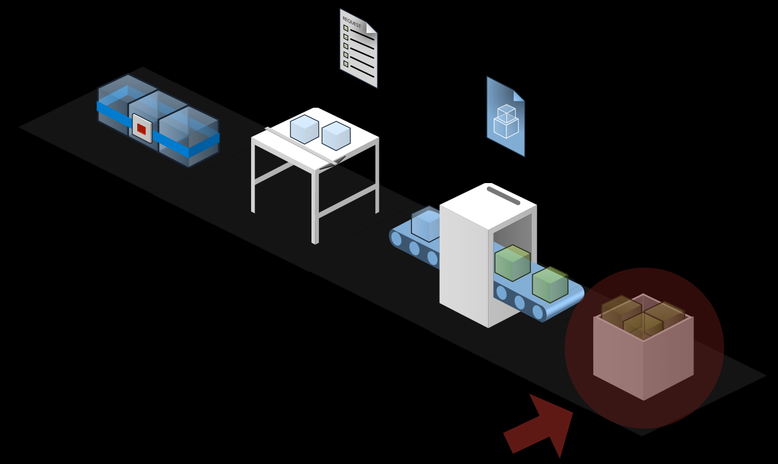
- Interaction through NSPersistentStoreCoordinator
- Stores created by type:
  - NSBinaryStoreType
  - NSXMLStoreType
  - NSSQLiteStoreType
  - NSInMemoryStoreType

Decision: Which Store Type?

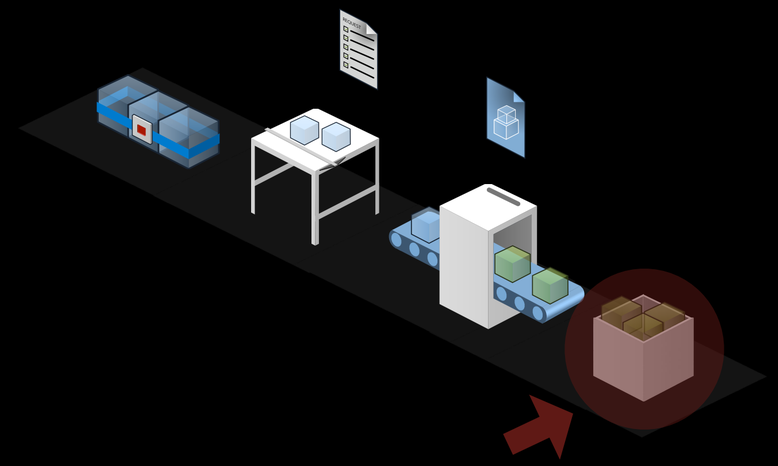
	XML	Binary	In Memory	SQLite
Speed	Slow	Fast	Fast	Fastest
Graph Size	Whole	Whole	Whole	Partial
Other	Externally parseable		No backing required	

In Memory Stores

- Just like any other store
- Doesn’t save to disk anywhere
- Useful for storing transient objects
  - singletons that are created every time

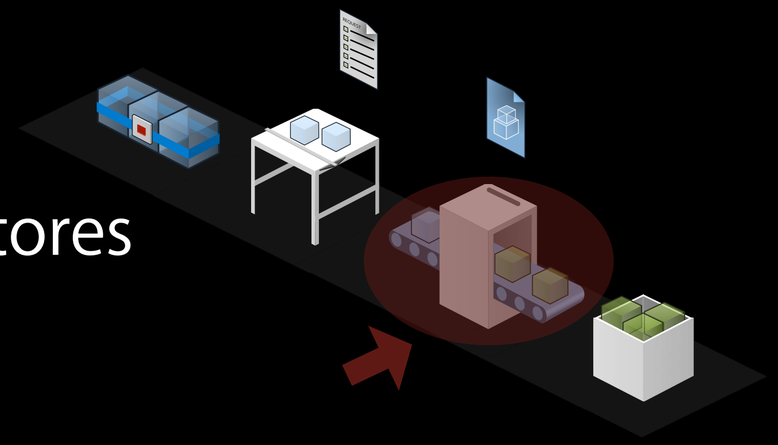


Problem: Legacy File Formats



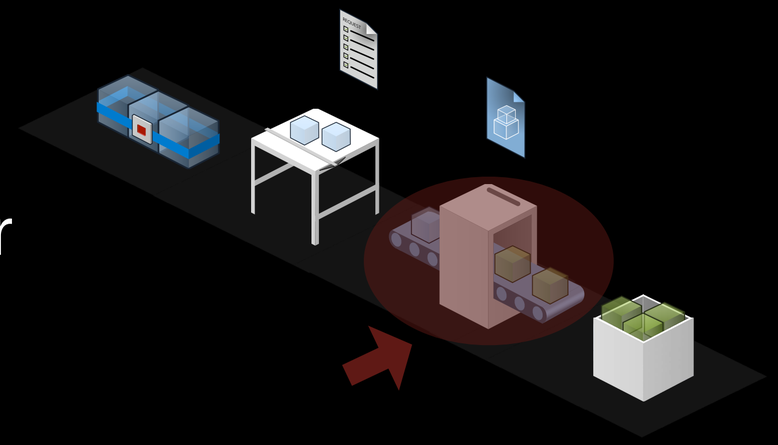
- Create your stack backed by an NSInMemoryStoreType
- Read your file and create your instances of NSManagedObject
- Use assignObject:toPersistentStore: to associate your objects with the in memory store
- Register for NSManagedObjectContextDidSaveNotification
- During objects associated with the store and manually save convert and save them

Working with Multiple Stores



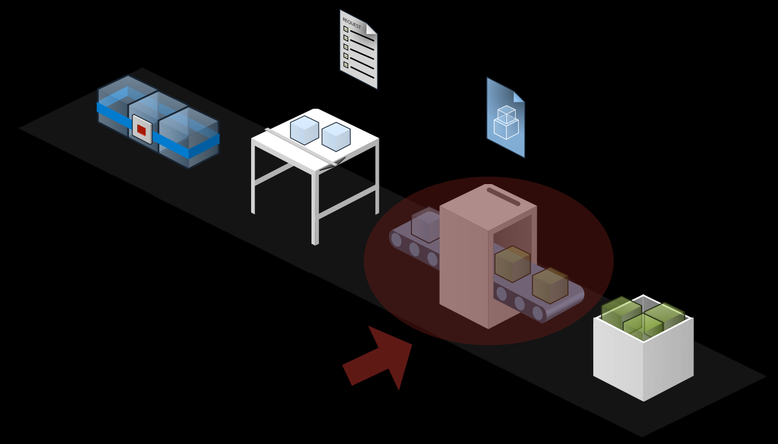
- By default, fetch requests retrieve data from all stores
- Use affectedStores accessors on fetch requests to direct requests to specific stores
- Objects are automatically assigned to stores at save time
  - Assignment can also be done using NSManagedObjectContext assignObject:toPersistentStore:
- Use configurations to relate entities to stores
- Considerations with Cocoa Bindings

Store Migration



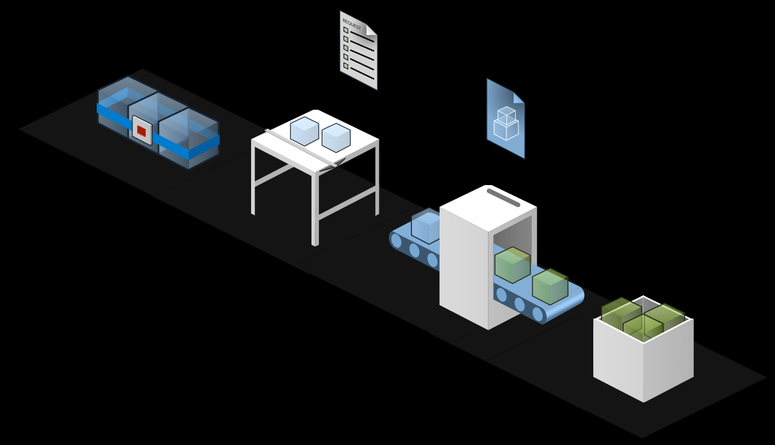
- Simple to move data from one format to another
- One call on NSPersistentStoreCoordinator
- Some caveats
  - Works by pulling everything in store into memory and resaving
  - Time and memory expensive
  - Old store will be removed
    - Managed Object Context will transition all its managed objects to new store
    - But objectIDs referencing the old store will be invalid

Store Migration

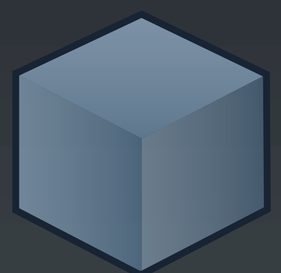


```
[coordinator migratePersistentStore: aStore toURL: aURL options: nil
withType: NSSQLiteStoreType error: &error];
```

Decision: NSPersistentDocument or Not



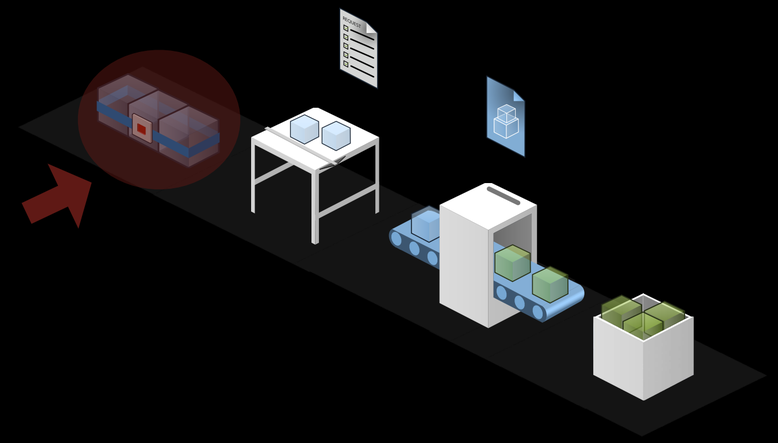
- Use NSPersistentDocument if
  - Data is contained in one store
  - Don’t have to worry about data from other sources
- Otherwise you want a non-document-based application



Cross-store Relationships

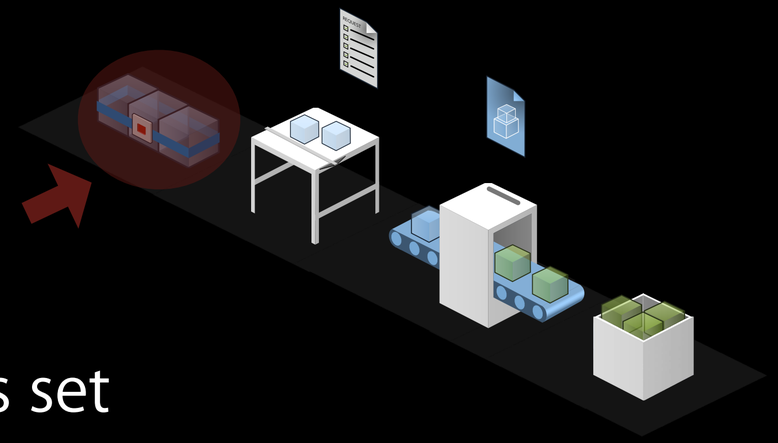
http://developer.apple.com/wwdc2005/

Creating Cross Store Relationships



- Not supported out of the box
- Can be implemented using
  - Transient properties
  - Fetched properties
- Requires a custom subclass of NSManagedObject

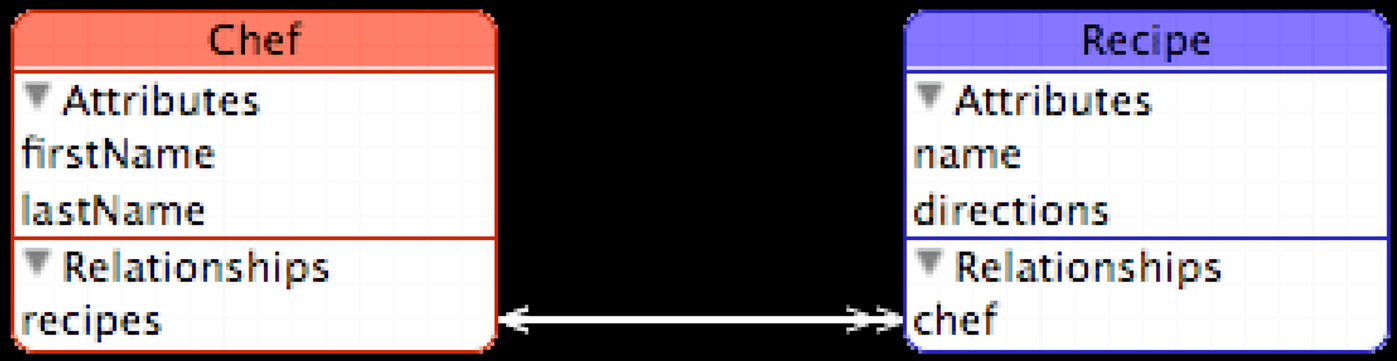
Two Patterns



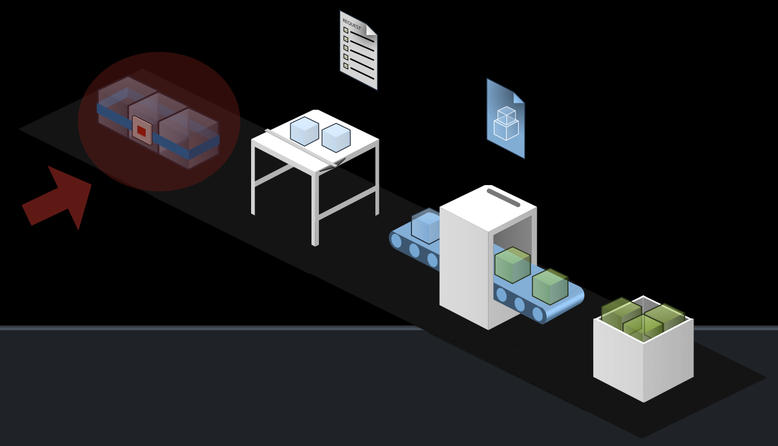
- Update storage at set time
  - Override custom accessor to push data to persistent backing attribute when new value is set
- Update storage at save time
  - Override -willSave to push data to persistent backing attribute
- Resolve relationships in awakeFromFetch

Cross Store Example

The Model

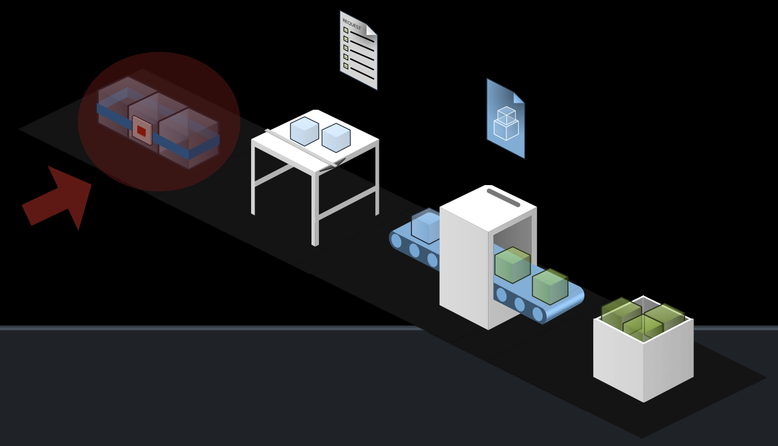


Code Example



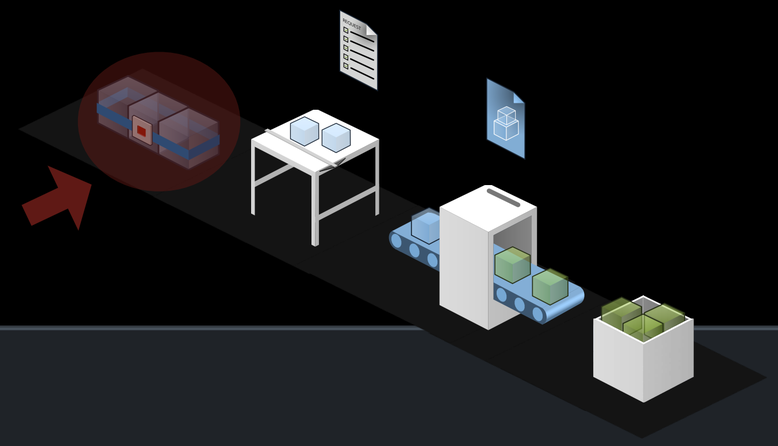
```
(void)willSave {
  [super willSave];
  [self willAccessValueForKey: @”chef”];
  NSManagedObjectID *moid = [[self primitiveValueForKey: @”chef”] managedObjectID];
  [self setPrimitiveValue: [moid URIRepresentation] forKey: @”chefid”];
  [self didAccessValueForKey: @”chef”];
}
```

Code Example



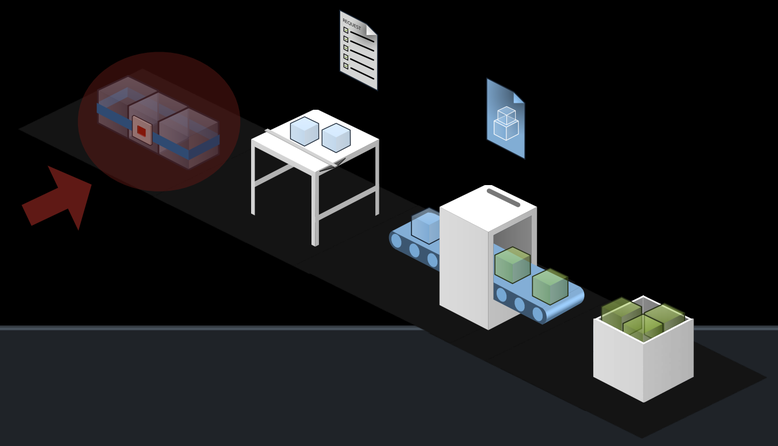
```
(void)willSave {
  [super willSave];
  [self willAccessValueForKey: @”chef”];
  NSManagedObjectID *moid = [[self primitiveValueForKey: @”chef”] managedObjectID];
  [self setPrimitiveValue: [moid URIRepresentation] forKey: @”chefid”];
  [self didAccessValueForKey: @”chef”];
}
```

Code Example



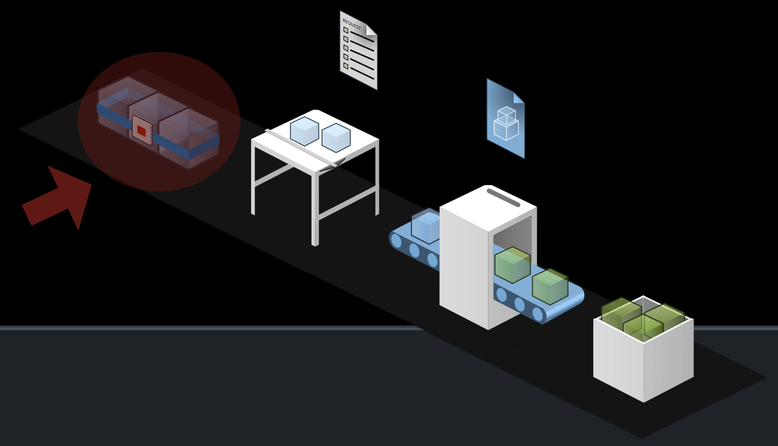
```
(void)willSave {
  [super willSave];
  [self willAccessValueForKey: @”chef”];
  NSManagedObjectID *moid = [[self primitiveValueForKey: @”chef”] managedObjectID];
  [self setPrimitiveValue: [moid URIRepresentation] forKey: @”chefid”];
  [self didAccessValueForKey: @”chef”];
}
```

Code Example



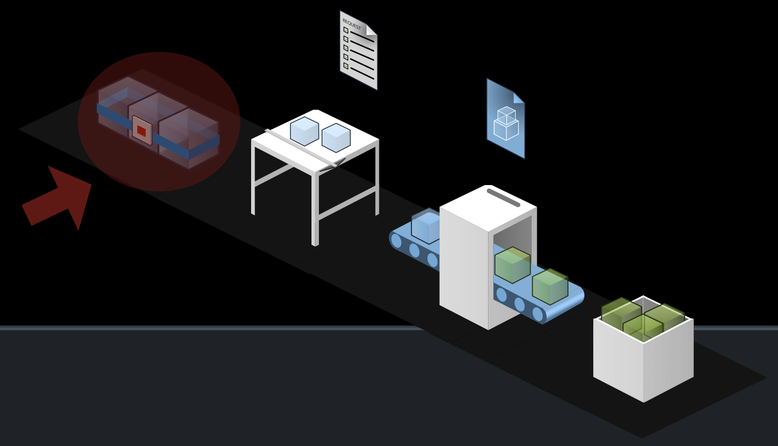
```
(void)willSave {
  [super willSave];
  [self willAccessValueForKey: @”chef”];
  NSManagedObjectID *moid = [[self primitiveValueForKey: @”chef”] managedObjectID];
  [self setPrimitiveValue: [moid URIRepresentation] forKey: @”chefid”];
  [self didAccessValueForKey: @”chef”];
}
```

Code Example



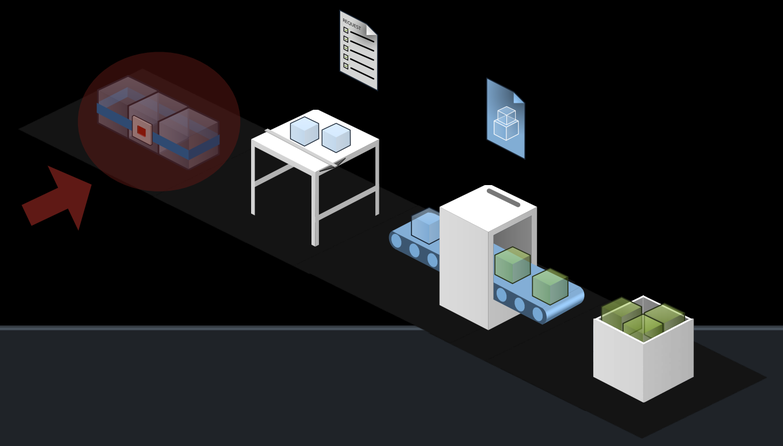
```
(void)willSave {
  [super willSave];
  [self willAccessValueForKey: @”chef”];
  NSManagedObjectID *moid = [[self primitiveValueForKey: @”chef”] managedObjectID];
  [self setPrimitiveValue: [moid URIRepresentation] forKey: @”chefid”];
  [self didAccessValueForKey: @”chef”];
}
```

Code Example



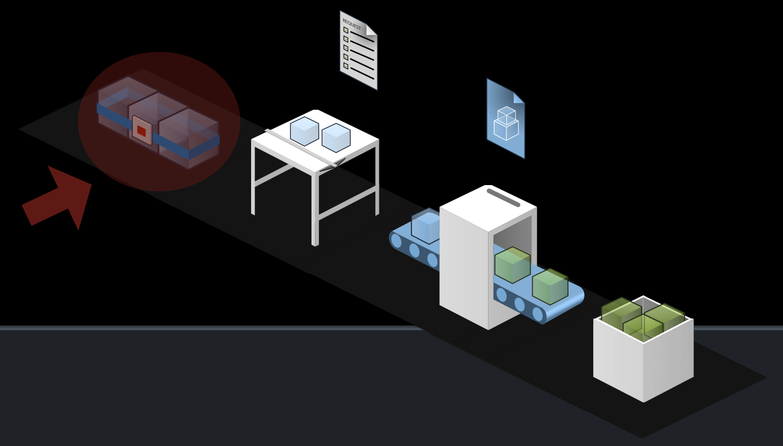
```
(void)willSave {
  [super willSave];
  [self willAccessValueForKey: @”chef”];
  NSManagedObjectID *moid = [[self primitiveValueForKey: @”chef”] managedObjectID];
  [self setPrimitiveValue: [moid URIRepresentation] forKey: @”chefid”];
  [self didAccessValueForKey: @”chef”];
}
```

Code Example



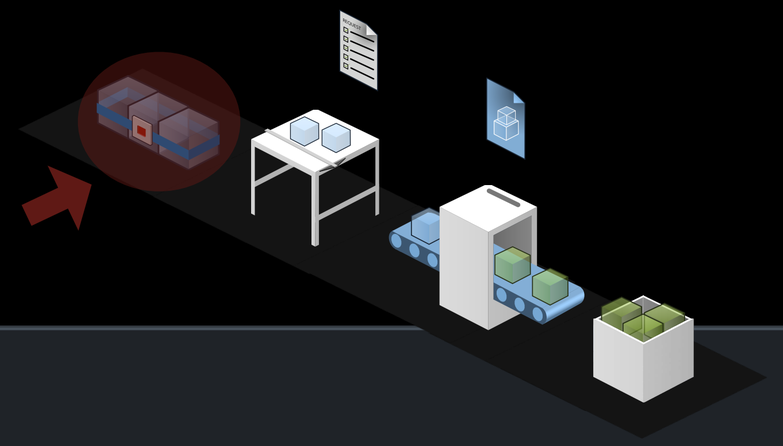
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



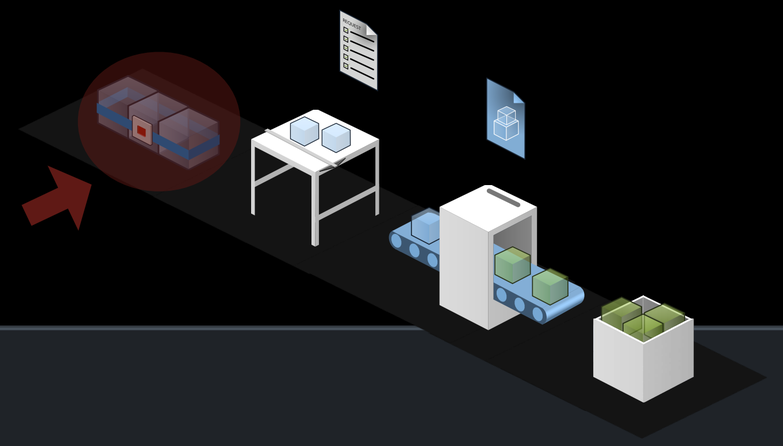
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



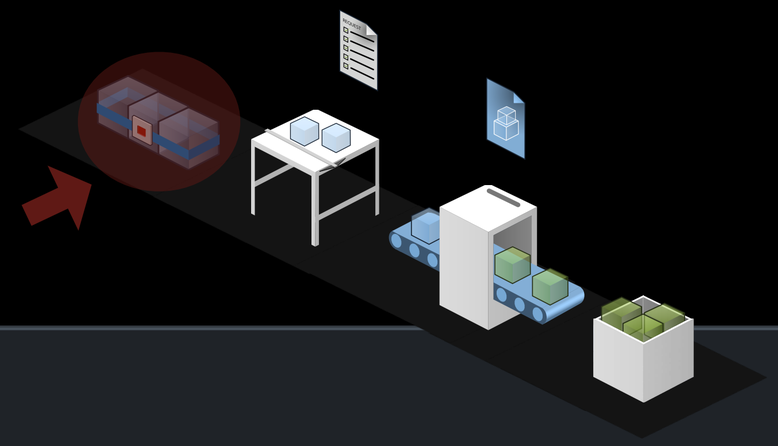
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



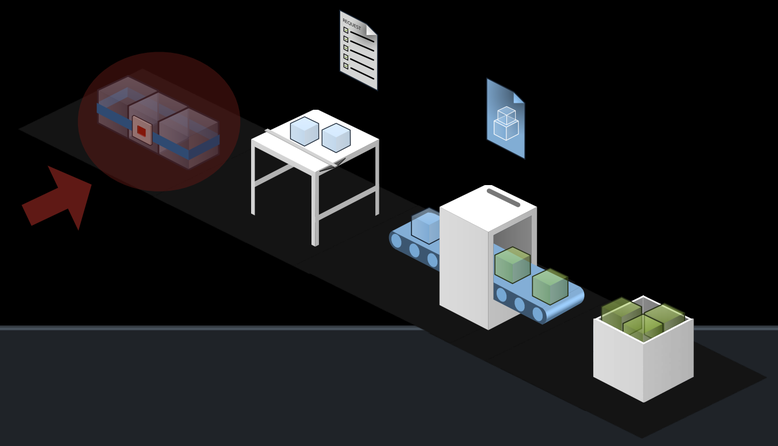
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



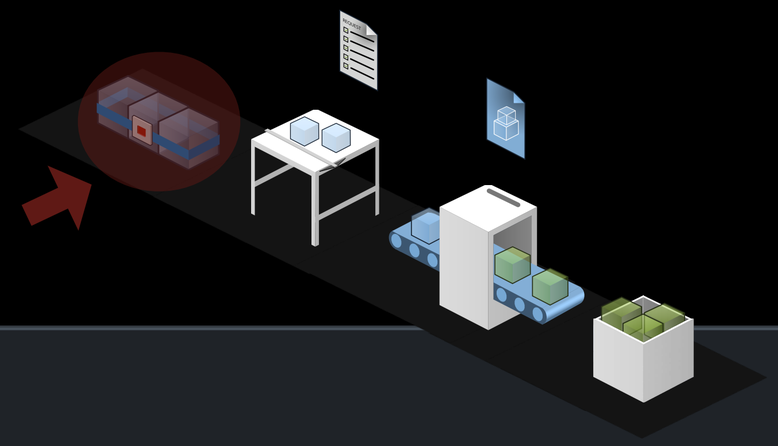
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



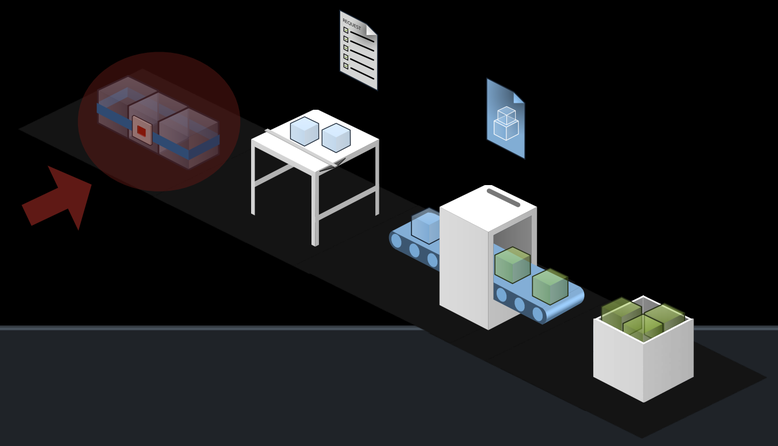
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Code Example



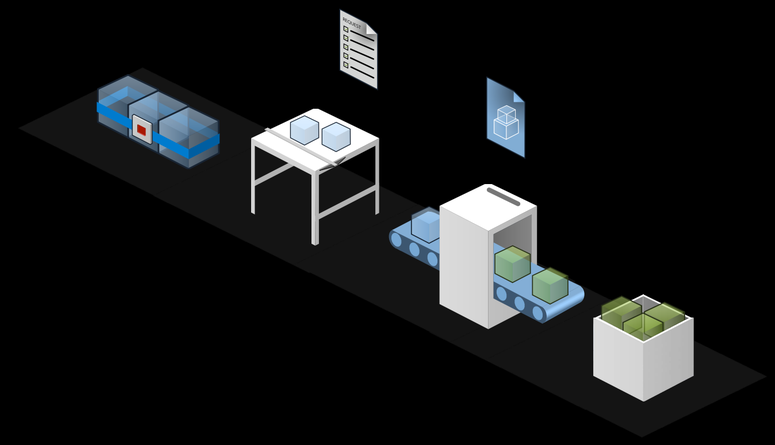
```
- (void)awakeFromFetch {
  [super awakeFromFetch];
  NSURL *chefiduri = [self primitiveValueForKey: @”chefid”];
  NSManagedObjectContext *context = [self managedObjectContext];
  NSPersistentStoreCoordinator *coordinator = [context persistentStoreCoordinator];
  NSManagedObjectID *moid = [coordinator managedObjectIDForURIRepresentation:
chefiduri];
  NSManagedObject *localChef = [context objectWithID: moid];
  [self setPrimitiveValue: localChef forKey: @”chef”];
}
```

Advanced Topics

http://developer.apple.com/wwdc2005/

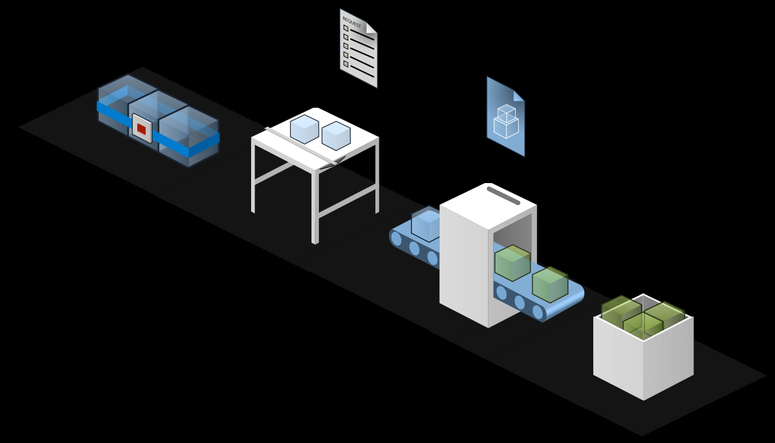
Spotlight Integration

Small Amounts of Metadata



- Put information you want exported into store metadata
- Use metadataForPersistentStoreWithURI:error: in Spotlight importer to extract data
- Duplicate data increases store size
- Recommended approach
  - Faster

Larger Amounts of Metadata

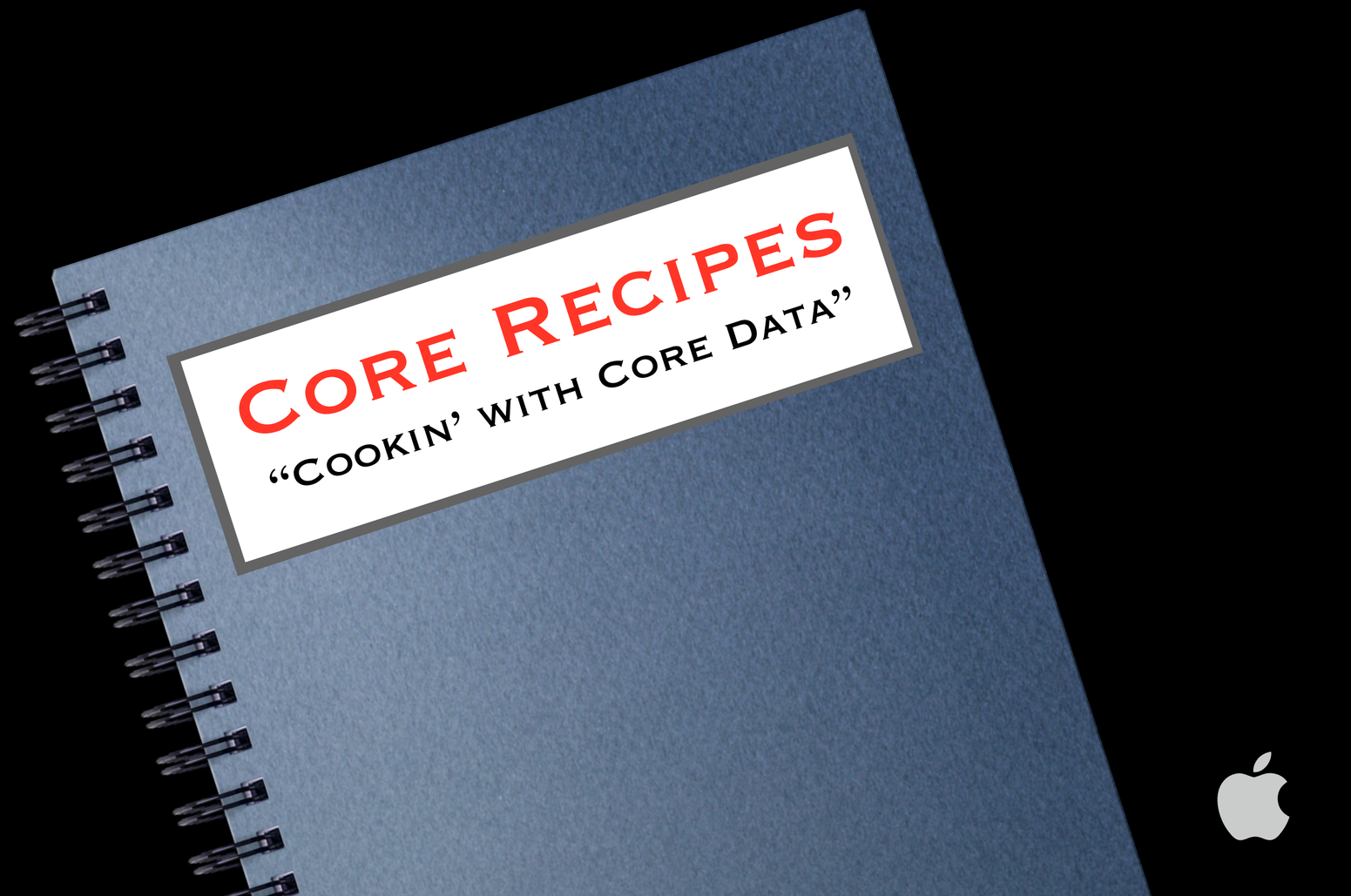


- Store information as you normally would
- Create a persistent stack in Spotlight importer
- Use fetch requests to retrieve desired data
- Persistent stacks are heavyweight
- Slower, but no duplication

WWDC Sample Code

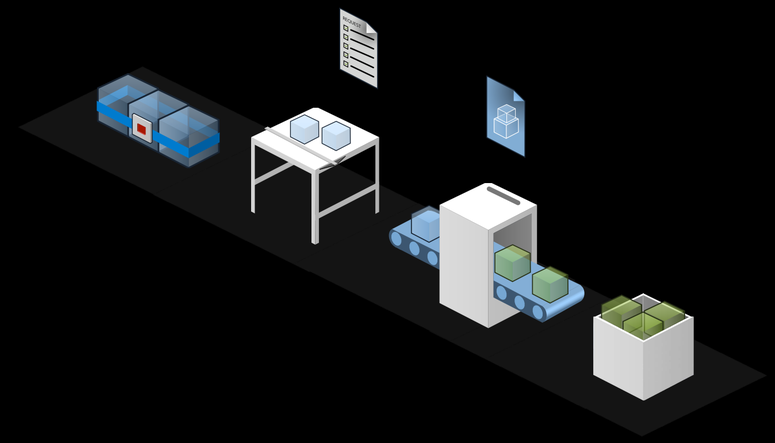
“CoreRecipes”

- CoreRecipes application
  - Spotlight integration



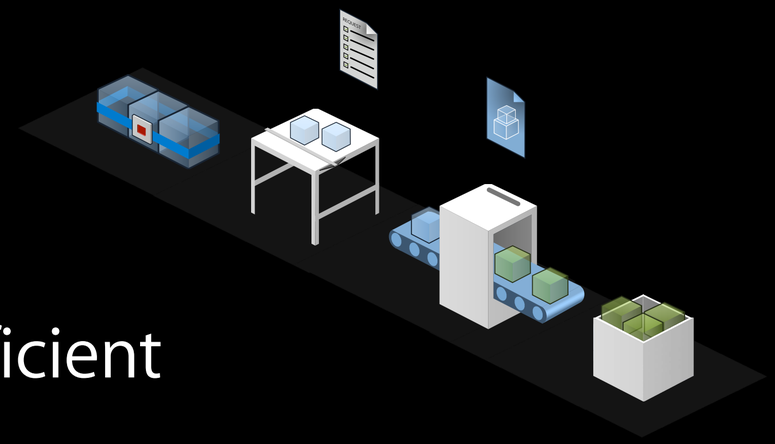
Memory Management

Memory Management



- Weak references between MOs and MOC
- Managed Objects can act as proxies
- Faulting is similar to virtual memory paging
- Core Data Programming Guide
  - Section “Using Managed Objects”

Problem: How to Handle Large Datasets

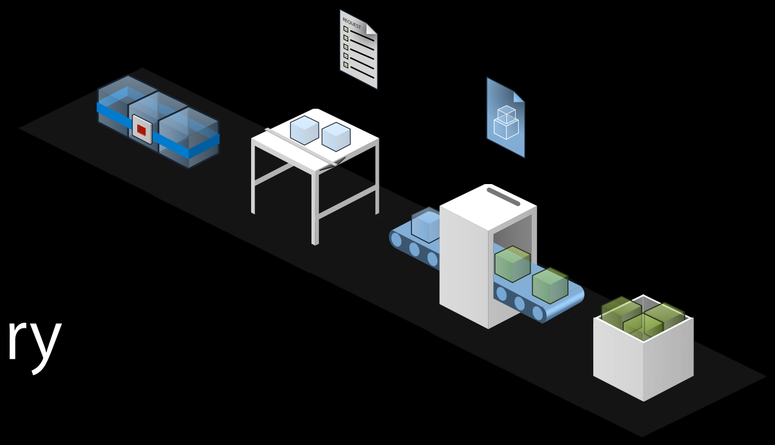


- Opening atomic stores is slow and memory inefficient
  - Use SQLite store if possible
  - Binary is the second best option
- Minimize working set
  - Only retain objects you need
  - Let others be turned back into faults and/or released

Threads

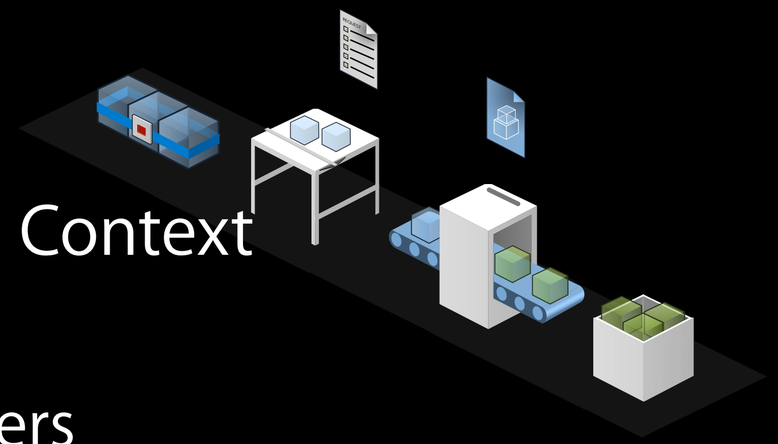
http://developer.apple.com/wwdc2005/

Threading Basics



- CoreData classes not implicitly thread safe
- NSManagedObject similar to NSMutableDictionary
- CoreData framework locks objects it uses
  - Context locks Coordinator internally during fetch, save, etc
- You must lock (or not share) objects you use
  - Generally “use” means:
    - send messages to an object
    - access an object’s ivars

Threading Tips



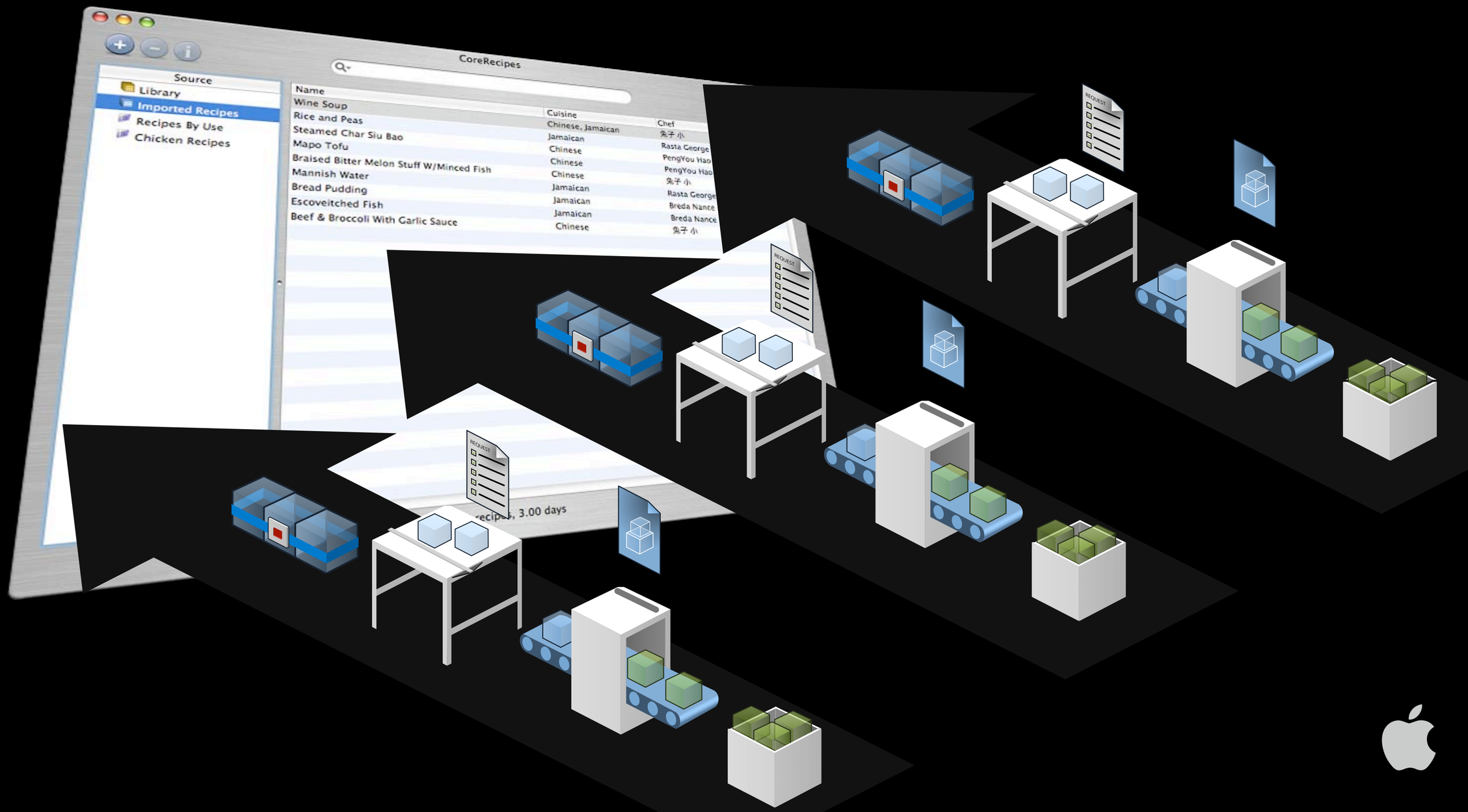
- Each thread has its own private Managed Object Context
- Only pass managed object IDs!
  - Okay to pass objectIDs between threads as pointers
  - Unnecessary to use URIRepresentation
- Do not pass managed objects or contexts between threads

One Context per Thread

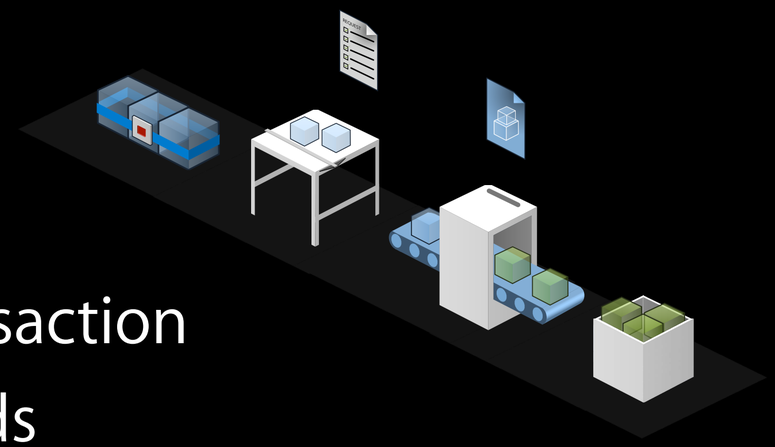
Shared Persistent Store Coordinator

One Stack per Thread

More concurrency, but no shared caches

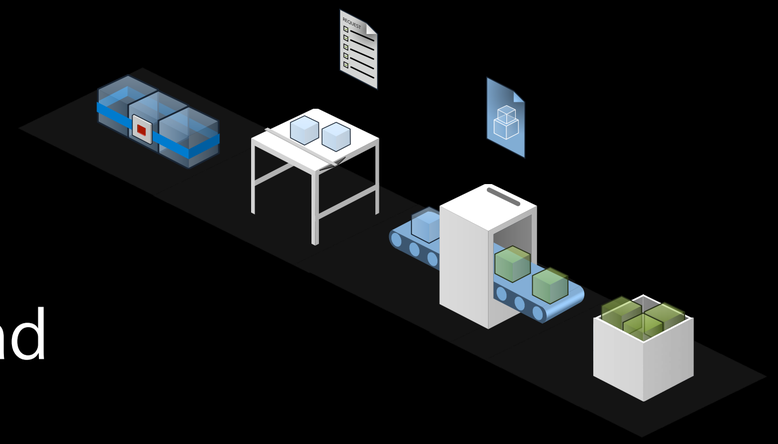


Locking



- Not just thread safety
  - Locking signals to framework the scope of a transaction
  - Prevents interspersing of edits by different threads
  - Save changes coherently
  - Keep operations distinct on the undo stack
- Therefore
  - Thread must own Context to access any Managed Object registered with that Context
  - Locking around a Managed Object setter insufficient
    - Lock the Context instead

Cocoa Bindings & AppKit

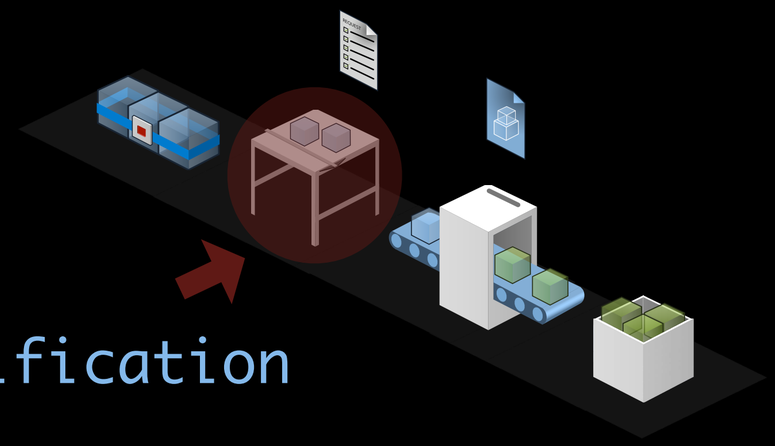


- Completely totally NOT thread safe
- All work with Controllers MUST be on main thread
- Use -performSelectorOnMainThread:

Core Data Notifications

http://developer.apple.com/wwdc2005/

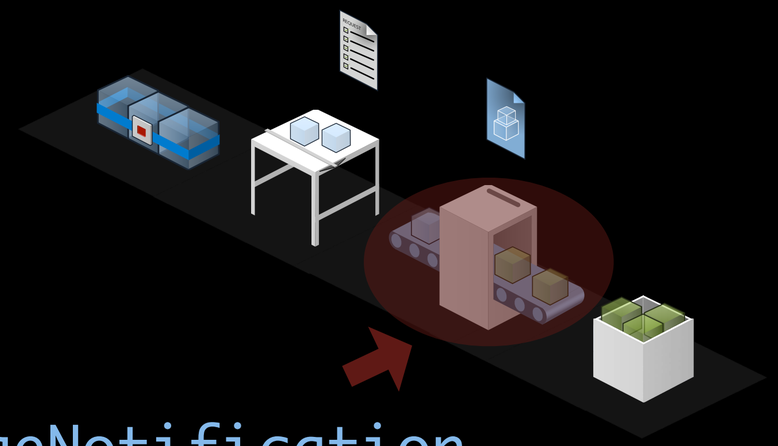
MOC Notifications



- `NSManagedObjectContextObjectsDidChangeNotification`
- `NSManagedObjectContextDidSaveNotification`

- Notification userInfo dictionary has
  - Inserted, updated, deleted keys
  - NSSets of managed objects

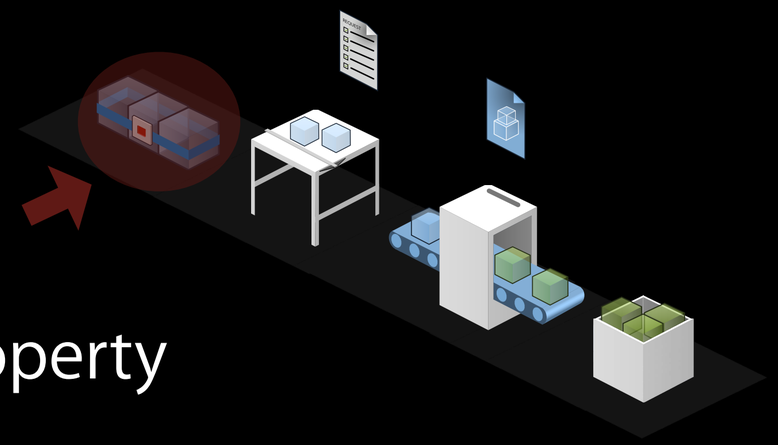
PersistentStoreCoordinator Notifications



- `NSPersistentStoreCoordinatorStoresDidChangeNotification`

- Notification userInfo dictionary has
  - Added, removed, UUID changed keys
  - NSArrays of persistent stores references

KeyValueObserving Notifications



- Managed Objects send KVO notifications per property
  - will/did access for valueForKey: or public getters
  - will/did change for setValue:forKey: or public setters
  - Undo and redo operations
  - Relationship maintenance (inverses, delete propagation)
- Primitive KVC operations do NOT send KVO notifications

Common Errors

http://developer.apple.com/wwdc2005/

Developer Errors

- Changing a model and accessing previously saved data
- Forgetting to send KVO notifications in custom accessors
  - Accessing ivars directly
  - Misusing primitiveValueForKey: & setPrimitiveValueForKey:
- Not letting the framework drive
  - Retaining objects you don’t need
  - Implementing accessors you don’t need
  - Micromanaging

Common Error Messages

```
NSError with reason @”Could not merge changes.”
```

- Two writes conflicted (different contexts tried to change the same data) aka optimistic locking failure
- Set a merge policy on the context, or
- Manually resolve the failure

Common Error Messages

```
NSError with reason @”Core Data could not
fulfill a fault.”
```

- The object’s underlying data has been deleted from the object store
- You should discard this object

Common Exceptions

```
*** Uncaught exception:
<NSUnknownKeyException> [<MyMO 0x3036b0>
valueForUndefinedKey:]: this class is not key
value coding-compliant for the key randomKey.
```

- You have a typo in the key, or
- You initialized your managed object with -init instead of initWithEntity:inManagedObjectContext:

Common Exceptions

```
*** Uncaught exception:
<NSInvalidArgumentException> [<MyMO
0x3036b0>_assignObject:toPersistentStore:]: Can’t
assign an object to a store that does not contain
the object’s entity.
```

- You are trying to assign an object to a store that can’t hold it
- Assign object to a store which contains instances of the objects’ entity

Common Exceptions

```
*** Uncaught exception:
<NSInvalidArgumentException> [<MyMO
0x3036b0>_assignObject:toPersistentStore:]: Can’t
reassign an object to a different store once it has
been saved.
```

- The object you are trying to assign has already been assigned and saved
- To move it, you must create a new instance, copy the information from the old object, and delete the old instance

Common Exceptions

```
*** Uncaught exception:
<NSObjectInaccessibleException> [<MyMO
0x3036b0>_assignObject:toPersistentStore:]: The
NSManagedObject with ID:#### has been invalidated.
```

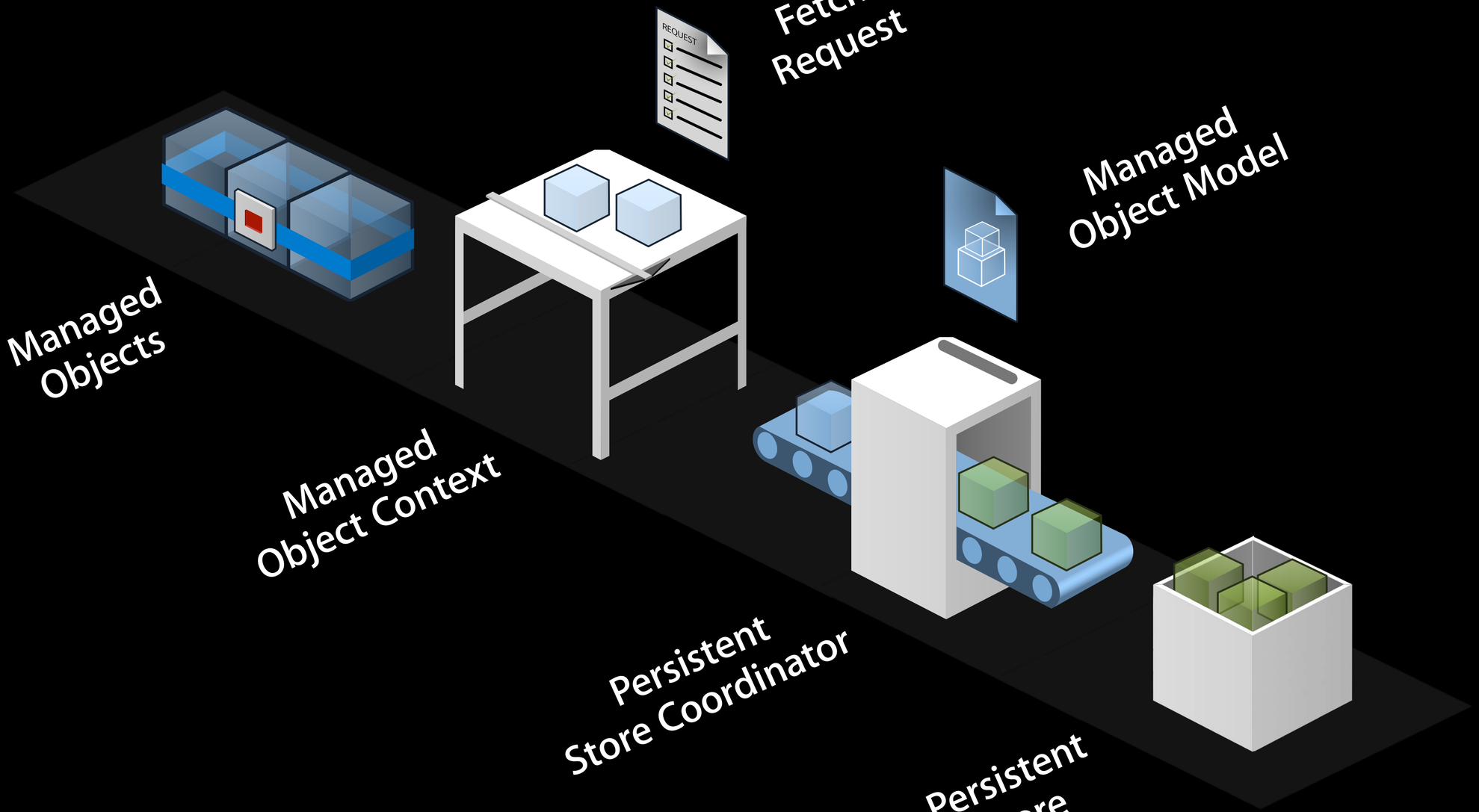
- The store for the fault you are attempting to fire has been removed, or
- The context has been -reset
- You should toss the object
- If you wish to re-add the store, fetch it again

Summary

http://developer.apple.com/wwdc2005/

Core Data

Object management and persistence



Summary

- Core Data architecture
- Roles of components in a Core Data stack
- Decision points when writing a Core Data application
- Patterns to use when confronted with complex problems
- Common errors and exceptions in Core Data applications

More Information

http://developer.apple.com/wwdc2005/

Documentation, Sample Code, and Other Resources

http://developer.apple.com/wwdc2005/

Already at Your Fingertips…

Lots of information and samples

Documentation

- Core Data Programming Guide
- NSPersistentDocument Core Data Tutorial
- Low-Level Core Data Tutorial
- Core Data Objective-C Reference (and update)

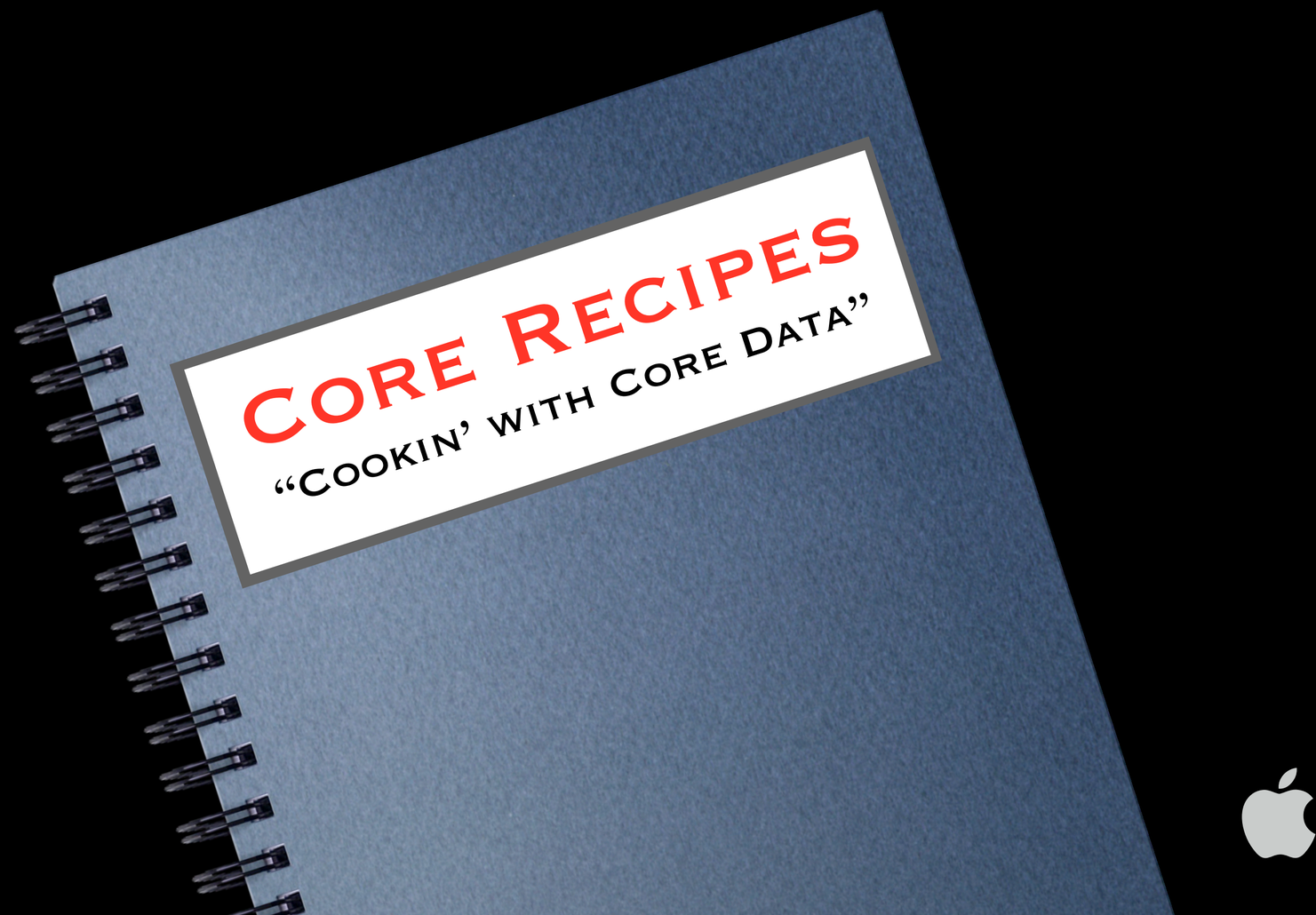
/Developer/Examples/CoreData

- OutlineEdit
- Stickies
- Tile Puzzle
- Event Manager

WWDC Sample Code

“CoreRecipes”

- Hands-on samples (for tomorrow’s session)
  - Command-line and GUI application
- CoreRecipes application
  - Core Data essentials
  - Cocoa bindings
  - Spotlight integration
  - Automator action
- Data migration sample



Who to Contact

Matthew Formica
Cocoa/Core Data Evangelist
mformica@apple.com

Cocoa Development
Apple Email Discussion List
cocoa-dev@lists.apple.com

Bug Reporter
Apple Bug Tracking System
http://bugreport.apple.com

Q&A

http://developer.apple.com/wwdc2005/

WWDC2005

Worldwide Developers Conference